HAL
open science

Home Devices Mediation using ontology alignment and code generation techniques

Charbel El Kaed

To cite this version:

Charbel El Kaed. Home Devices Mediation using ontology alignment and code generation techniques. Ubiquitous Computing. Université de Grenoble, 2012. English. NNT : 2012GRENM002 . tel-00680022

HAL Id: tel-00680022

https://theses.hal.science/tel-00680022v1

Submitted on 17 Mar 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université de Grenoble

THÈSE

Pour obtenir le grade de

Docteur de l'Université de Grenoble

Spécialité : INFORMATIQUE

Arrêté ministériel : 7 août 2006

Présentée et soutenue publiquement par

CHARBEL EL KAED

13 Janvier 2012

Home Devices Mediation using ontology alignment and code generation techniques

---

Thèse dirigée par YVES DENNEULIN et codirigée par FRANÇOIS-GAËL OTTOGALLI

Jury

Stéphane	FRÉNOT	Professeur, INSA Lyon	Rapporteur
Michel	RIVEILL	Professeur, Polytech'Nice Sophia	Rapporteur
Patrick	REIGNIER	Professeur, Grenoble INP	Examinateur
Yves	DENNEULIN	Professeur, Grenoble INP	Encadrant
F.G.	OTTOGALLI	Responsable de Recherche, Orange Labs	Examinateur
Łukasz	SZÒSTEK	Expert TP R&D, Orange Labs, Poland	Invité

Thèse préparée au sein d'Orange Labs et du Laboratoire d'Informatique de Grenoble, dans l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique

Abstract

Ubiquitous systems imagined by *Mark Weiser* are emerging thanks to the development of embedded systems and plug-n-play protocols like the Universal Plug aNd Play (UPnP), the Intelligent Grouping and Resource Sharing (IGRS), the Device Profile for Web Services (DPWS) and Apple Bonjour. Such protocols follow the service oriented architecture (SOA) paradigm and allow an automatic device and service discovery in a home network.

Once devices are connected to the local network, applications deployed for example on a smart phone, a PC or a home gateway, discover the plug-n-play devices and act as control points. The aim of such applications is to orchestrate the interactions between the devices such as lights, TVs and printers, and their corresponding hosted services to accomplish a specific human daily task like printing a document or dimming a light.

Devices supporting a plug-n-play protocol announce their hosted services each in its own description format and data content. Even similar devices supporting the same services represent their capabilities in a different representation format and content. Such heterogeneity along with the protocols layers diversity, prevent applications to use any available equivalent device on the network to accomplish a specific task. For instance, a UPnP printing application cannot interacts with an available DPWS printer on the network to print a document.

Designing applications to support multiple protocols is time consuming since developers must implement the interaction with each device profile and its own data description. Additionally, the deployed application must use multiple protocols stacks to interact with the device. More over, application vendors and telecoms operators need to orchestrate devices through a common application layer, independently from the protocol layers and the device description.

To accomplish interoperability between plug-n-play devices and applications, we propose a generic approach which consists in automatically generating proxies based on an ontology alignment. The alignment contains the correspondences between two equivalent devices descriptions. Such correspondences actually represent the proxy behavior which is used to provide interoperability between an application and a plug and play device. For instance, the generated proxy will announce itself on the network as a UPnP standard printer and will control the DPWS printer. Consequently, the UPnP printing application will interact transparently with the generated proxy which adapts and transfers the invocations to the real DPWS printer.

We implemented a prototype as a proof of concept that we evaluated on several real UPnP and DPWS equivalent devices.

Acknowledgements

First, I would like to thank the jury members, especially Stéphane Frénot and Michel Riveill for evaluating my dissertation and for their interesting feedbacks. I also thank Patrick Reignier for accepting to chair the defense session, and Łukasz Szòstek for examining my work.

I express my gratitude to my advisors, Yves Denneulin and François-Gaël Ottogalli for their support and trust over the years. And specially for giving me the freedom to explore areas of research which meet my interest.

In addition I thank Roland Airiau, Nordine Oulahal and Vincent Olive, along with Sébastien Bolle and Serge Martin for giving me the opportunity to be involved in the MADE team and its several projects.

Furthermore, it was a real pleasure to work with the MADE team, a very special "Thank You" goes to Loïc Petit, Julien Rouland, Stéphane Séyvoz, Maxime Louvel, Radu Kopetz, Xavier Roubaud, Rémi Melisson, Rémi Drhuile, Jacques Pulou and Mathieu Anne for their support and for the special time we had talking about technical, geeky, economical, political and science-fiction based systems. I believe that all these moments have shaped the coffee breaks into an amazing time. I am also grateful for the implementation effort carried out by the interns, Felipe Melo and Thierry Sabran. I am also indebted to Sylvain Marié for his kind support during the integration of the DPWS stack. A special acknowledgment goes also to the members of the MESCAL and MOAIS teams for their welcome.

I have been blessed with numerous people motivating and encouraging me all these years, they can rejoice, their prayers have been heard. I thank the "FEU" for their warm welcome and support, André, Anna, Ben, Chris, David^n, Elke, François, Laurent, Léonce, Lucie, Maxime, Myriam^n, Nicolas^n, Roula, Simon, Sylvain and the rest of the large "FEU" family members. A special thank you goes to Priscilla for all her encouraging words.

I also wish to thank Ahmad^2, Hassan, Imad, Joe, Mazen and Wissam for these wonderfull years growing up together away from home. It also gives me great pleasure in acknowledging the support of Tesnim through all our university studies, thank you for your friendship all these 8 years. It was a great pleasure for me to accomplish a part of my ambitions next to you all.

I would like to thank some special friends who contributed in a way or another from behind the sea, Alex, Assaf, Bassam, Bernard, Charbel, Christelle, Cyntia, Elie, Jean, Jerome, Joe, Joanna, Layal, Rawad, Roger, Rony, Sabine, and many others not cited here...

I am indebted to my parents Joseph and Nouhad along with my sister Cherine for their unconditional love, and their patience while I was away from them. A special thank also goes to my relatives for their support, I thank the Bardawils, Chalach, Kaeds, Mansouras, Sabas.

"I can do all things through Him who strengthens me."

– Paul the Apostle

Contents

1 Introduction **19**

1.1 Problem Statement . . . 19

1.2 Contributions . . . 20

1.3 Thesis Outline . . . 22

I Context **25**

2 Ubiquitous Computing **27**

2.1 Ubiquitous Scenario . . . 27

2.2 Ubiquitous Environment Characteristics . . . 32

2.2.1 Dynamicity . . . 32

2.2.2 Heterogeneity . . . 33

2.3 Ubiquitous System Characteristics and Challenges . . . 35

2.3.1 Discovery . . . 36

2.3.2 Control . . . 37

2.3.3 Eventing . . . 37

2.3.4 Interoperability . . . 37

2.3.5 Inference . . . 38

2.3.6 Interpretation . . . 38

2.3.7 Security . . . 38

2.4 The Digital Home Towards a Ubiquitous Environment . . . 38

2.4.1 Discovery and Adaptation for Dynamicity . . . 39

2.4.2 Interoperability for Heterogeneous Plug and Play Protocols . . . 40

2.4.3 Management for Devices and Applications . . . 40

2.4.4 The Overall Actual Device and Application Architecture . . . 42

2.4.5 Discussion Around the Ubiquitous System Characteristics . . . 43

2.5 Conclusion . . . 44

3 Service Oriented Architecture **47**

3.1 SOA Principles . . . 47

3.2 SOA Characteristics meeting Ubiquitous System Ones . . . 49

3.3 OSGi a Service Oriented Framework . . . 50

3.3.1 An Architectural Overview of OSGi . . . 51

3.3.2 Base Drivers . . . 55

3.4 Conclusion . . . 56

4 Plug and Play Protocols **57**
4.1 Plug-and-Play . . . 57
4.2 Common features . . . 58
4.3 Universal Plug and Play Protocol . . . 59
4.4 Device Profile for Web Services . . . 63
4.5 Intelligent Grouping and Resource Sharing . . . 66
4.6 Bonjour . . . 67
4.7 Plug and Play Protocols Divergence . . . 68
4.8 Conclusion . . . 70

5 Knowledge Representation **73**
5.1 Ontologies . . . 75
5.1.1 Ontology Entities . . . 76
5.1.2 Ontology Development Methodologies . . . 76
5.1.3 Semantic Web Services . . . 77
5.2 Rules . . . 79
5.2.1 Logic-Based Representation . . . 79
5.2.2 Rule Languages . . . 80
5.3 Models . . . 81
5.4 Conclusion . . . 82

6 Conclusion & Problem Statement: Device Interoperability **85**

II Related Work **89**

7 Overview of the Interoperability Frameworks **91**
7.1 Common Ontology . . . 92
7.1.1 Paolucci's Semantic Matching Algorithm . . . 92
7.1.2 PERSE: PERvasive SEmantic-aware Middleware . . . 93
7.1.3 MySIM . . . 93
7.2 Abstract Model . . . 95
7.2.1 DOG: Domotic OSGi Gateway . . . 95
7.2.2 EnTiMid . . . 96
7.2.3 PervML: Pervasive Modeling Language . . . 97
7.3 Uniform Language/Interface . . . 98
7.3.1 HomeSOA . . . 98
7.3.2 UMB: Universal Middleware Bridge . . . 99
7.3.3 DomoNet: Domotic Network . . . 100
7.4 Comparison & Discussion . . . 101

8 Ontology Matching **107**
8.1 Matching Techniques . . . 109
8.2 Ontology Alignment Tools & Frameworks . . . 112
8.3 Conclusion . . . 114

III Contribution **115**

9 Dynamic Service Adaptation for Devices' Interoperability **117**
9.1 Motivation . . . 117
9.2 Overview . . . 119
9.3 An End To End Architecture . . . 122
9.3.1 OWL Writers . . . 122
9.3.2 Overview of the Device Matching . . . 125
9.3.3 DOXEN . . . 125
9.3.4 Global Architecture . . . 128
9.4 Device Matching . . . 130
9.4.1 Ontology Alignment . . . 132
9.4.2 Expert Alignment Validation . . . 137
9.4.3 Pattern Detection . . . 138
9.4.4 Expert Code Annotation . . . 149
9.5 Concluding Remarks . . . 152

10 Implementation **155**
10.1 OWL Writers . . . 155
10.1.1 Flattening the WSDL . . . 157
10.1.2 The "*Bonjour*" Exception . . . 158
10.2 ATOPAI . . . 159
10.2.1 Ontology Alignment . . . 160
10.2.2 Pattern Detection . . . 162
10.2.3 Expert Adaptation with ATOPAI . . . 164
10.3 DOXEN . . . 167
10.3.1 Ontology Visiting . . . 168
10.3.2 Code Generation . . . 169
10.3.3 Compiling and Bundle Generation . . . 171
10.3.4 DOXEN's Supported Capabilities . . . 173
10.4 Experimentations . . . 175

11 Evaluations **181**
11.1 OWL Writer . . . 181
11.1.1 Ontology Generation Time . . . 182
11.1.2 Annotated Information in the Generated Ontologies . . . 185
11.2 Device Matching . . . 186
11.2.1 SMOA++ . . . 186
11.2.2 Alignment . . . 187
11.3 DOXEN . . . 194
11.3.1 Proxy Generation . . . 194
11.3.2 Proxy Invocation . . . 194
11.4 Discussion . . . 196
11.5 Conclusion . . . 197

IV Conclusion **199**

Conclusion **201**

12 Conclusion **201**
12.1 Contributions . . . 202
12.2 Perspectives . . . 204
12.2.1 Machine Learning Based Alignment . . . 204
12.2.2 Device Composition . . . 204
12.2.3 Security & Privacy . . . 205
12.2.4 Adaptation Code . . . 205
12.2.5 DOXEN . . . 205

A Publications **207**

B Additional Examples and Figures **209**
B.1 An OWL Ontology Example . . . 209
B.2 A DPWS PrintTicket Element . . . 210
B.3 DPWS Ontology Generation Example . . . 212
B.4 UPnP Binary Light Generated Ontology in OWL . . . 212
B.5 An Alignment Result between a UPnP and DPWS Lights in Align format . . . 217
B.6 Screen Shots of the UPnP-Android Based Home Controller . . . 219

C Detailed Alignment Results **221**
C.1 SMOA Printers Alignment Results . . . 221
C.2 SMOA++ Printers Alignment Results . . . 225
C.3 SMOA++ Printers Alignment Results with a Similarity Propagation . . . 227

List of Figures

1.1 Steps of the approach . . . 21

2.1 Ubiquitous Environments . . . 28
2.2 Ubiquitous Work Environments . . . 29
2.3 Ideal Ubiquitous System Architecture . . . 36
2.4 Remote Administration through CWMP and UPnP-DM . . . 41
2.5 Overall Layers System Architecture . . . 42

3.1 Service Oriented Architecture paradigm . . . 48
3.2 Service Composition . . . 49
3.3 OSGi on residential gateways . . . 50
3.4 Multi Layer Architecture of an OSGi Framework [Richard S. Hall] . . . 51
3.5 OSGi Bundle . . . 52
3.6 OSGi LifeCycle State Diagram . . . 52
3.7 The Service Hooks . . . 55
3.8 OSGi Base Drivers . . . 56

4.1 UPnP Device Description Structure . . . 60
4.2 DPWS Device Description Structure . . . 63
4.3 Hierarchical Parameters in the WSDL Printer description . . . 64
4.4 DPWS Exchange Example [Microsoft 06] . . . 65
4.5 IGRS System Architecture . . . 67

5.1 Knowledge Representation Techniques . . . 74
5.2 An Ontology Example . . . 75
5.3 A Context Representation Ontology Example [Pierson 09] . . . 76
5.4 Model Driven Architecture, Generalized Layers Example . . . 82

6.1 Plug And Play Interoperability Layers . . . 86
6.2 A Common Application Layer . . . 87

7.1 A Common Ontology Example . . . 92
7.2 MySIM Middleware [Ibrahim 08] . . . 94
7.3 Partial DogOnt Fragment showing Dimmer Lamp and Switch Instances [Bonino 09] . . . 95
7.4 (a) EnTiMid Architecture (b) An EnTiMid-UPnP Model Mapping example [Nain 08] . . . 96
7.5 PervML Models [Munoz 04] . . . 97
7.6 The HomeSOA Architecture [Bottaro 08a] . . . 98

7.7 UMB Architecture and a Virtual Device Proxy information [Moon 05] 100
7.8 The DomoNet Architecture [Miori 06] 100

8.1 Integrated Approach 107
8.2 Federated Approach 108
8.3 An Alignment Example Between Two Ontologies [Euzenat 07] 109
8.4 An Internal Structure Alignment Example Between Two Ontologies [Euzenat 07] 111

9.1 UPnP as a Common Application Layer 119
9.2 Architectural View of the UPnP-DPWS Proxy 120
9.3 Overview Of The Approach 121
9.4 The Modular Architecture 122
9.5 Ontology Generation by the OWL Writers 123
9.6 M2 Layer, General Device Model 123
9.7 (Simplified) UPnP Ontology Generation from an XML description, (properties view) 124
9.8 Part of the Taxonomic Structure of a UPnP (left) and a DPWS (right) Light 124
9.9 Part of a Lights Ontology 125
9.10 DOXEN Diagram Generation 126
9.11 DOXEN 127
9.12 Global Architecture 128
9.13 Sequential Diagram showing DOXEN and the Proxy Interaction 129
9.14 The overall major steps of the process 130
9.15 The Device Matching Process Overview 131
9.16 Aligner Simple description 132
9.17 Part of a Basic Alignment Result Between Two Lights Ontology 132
9.18 SMOA++ Matching 134
9.19 An illustration of the similarity propagation 136
9.20 An illustration of the similarity enhancement 136
9.21 Step 2: Alignment Validation 137
9.22 ATOPAI snapshot 138
9.23 Part of the Alignment Result Between Two Lights Ontologies After Expert Validation 138
9.24 Step 3: Pattern Detection 139
9.25 A Part of a SMOA++ Alignment between UPnP and DPWS Clocks 139
9.26 N-to-M Union Mapping 143
9.27 Sequential Union detected on the Standard Printers 144
9.28 Example of a complex Sequential and Union Mappings 145
9.29 A possible order of the example in Figure 9.28 145
9.30 Step 4: Code Annotation 150
9.31 Adaptation Behavior: Use Case #1 151
9.32 Adaptation through External Service Invocation: Use Case #2 152

10.1 Generating Ontologies 156
10.2 The Bonjour Base Driver compared with other Plug and Play Base Drivers 158
10.3 ATOPAI Supported Features 159
10.4 ATOPAI: Meta Data Adaptation 164
10.5 ATOPAI: Adaptation Code 166

10.6 DOXEN Proxy Generation Architecture (Simplified) 170
10.7 Code Generation Dependency 171
10.8 Experimentation Devices and Topology 175
10.9 Presence Simulator application Scenario 176
10.10 UPnP-DPWS Proxies detected by the MyDevices Application 176
10.11 Basic Diagnostic Use Case 178

11.1 UPnP and DPWS OWL Writers Generation 183
11.2 UPnP Writer Generation 183
11.3 DPWS Writer Generation 184
11.4 Success rates with regard to several similarity propagation values 192
11.5 Real Printer Example 195

B.1 A Standard DPWS Print Ticket Element [Microsoft 07] 211
B.2 DPWS Ontology Generation from a WSDL description 212
B.3 UPnP Lights detected by the UPnP-Android based Home Controller Application 219
B.4 UPnP Printer detected by the UPnP-Android based Home Controller Application 219

List of Tables

4.1 UPnP Protocol Stack [UPnP 08] 59
4.2 DPWS Protocol Stack 65
4.3 An Apple Printer Supported Services Example [Apple 05] 68
4.4 Plug And Play Protocols Stack Comparison 69
4.5 Device Description Concepts Comparison 70

7.1 A Comparison between Interoperability Middlewares (✓:Supported, ≈: Partially Supported, χ: Not Supported, ?: Information Not Clearly Available) 103

9.1 Basic Matching Techniques Result with Synonyms and Antonyms 133
9.2 Part of a Mapping between a standard DPWS and a UPnP printer action 140
9.3 Decision table, (✓:success, x:fail, ?:undefined) 149
9.4 Equivalent actions for UPnP-DPWS Standard Printers 150

10.1 LoC of each module 175

11.1 Generated Ontologies 182
11.2 UPnP OWL Writer Evaluation of three size equivalent descriptions 184
11.3 Ontology Generation (LoC) per entity type 186
11.4 Comparison between basic Matching Techniques 186
11.5 Mapping between a DPWS and a UPnP light device 188
11.6 Mapping between a DPWS and a UPnP light device with Similarity Propagation 189
11.7 Mapping between a DPWS and a UPnP Clock device 190
11.8 Mapping between a DPWS and a UPnP Clock device with Similarity Propagation 190
11.9 Alignment Evaluation without Similarity Propagation 192
11.10 Patterns and Matching Concepts Detection Time 193
11.11 Device Matching: number of the matched entities 193
11.12 Device Matching: description size increase 193
11.13 Generated Proxy results on a PC 194
11.14 Generated Proxy results on a Sodaville STB 195
11.15 Printer action invocation Time (ms) 195
11.16 Completing Comparison of Table 7.1 197

C.1 Legend 221
C.2 SMOA Mapping between a DPWS and a UPnP Printer without Similarity Propagation 222
C.3 SMOA Mapping between a DPWS and a UPnP Printer (Continuation of Table C.2) 223
C.4 SMOA False Mapping between a DPWS and a UPnP Printer(Continuation of Table C.3) 224

C.5 SMOA++ Mapping between a DPWS and a UPnP Printer without Similarity Propagation . . . 225
C.6 SMOA++ Mapping between a DPWS and a UPnP Printer (Continuation of Table C.5) 226
C.7 SMOA++ False Mapping between a DPWS and a UPnP Printer (Continuation of Table C.6) . . 227
C.8 SMOA++ Mapping between a DPWS and a UPnP Printer with a Similarity Propagation 228

Listings

3.1 An OSGi Printer Service Example . . . 53
3.2 Printer Service Properties Example . . . 54
3.3 Printer Service Implementation Example . . . 54
4.1 Intel Binary Light Device Description (Simplified) . . . 60
4.2 UPnP SwitchPower Service Description (Simplified) . . . 61
4.3 DPWS SwitchPower Service Description (Simplified) . . . 64
4.4 LPR TXT record for a PostScript printer [Apple 05] . . . 68
5.1 RDF Example . . . 77
5.2 SPARQL Query Example . . . 80
5.3 OPPL Query Example . . . 81
7.1 Paolucci's four matching degrees . . . 92
7.2 Printing Service Description Example in MySIM . . . 94
8.1 A Fragment of WordNet 2.1 Results for the word "printer" . . . 110
9.1 The OPPL rule applied to detect the Simple_Mapping_Input Pattern . . . 141
9.2 The OPPL rule applied to detect the Union_1_to_n_Input Pattern . . . 142
9.3 An OPPL rule applied to detect the has_Next Pattern in general . . . 145
9.4 The OPPL Cycle detection Rule . . . 145
9.5 An OPPL rule applied to detect a Sequential Union Pattern . . . 146
9.6 A High Level Adaptation Code . . . 151
9.7 Adaptation through External Service Invocation . . . 152
10.1 A UPnP OWL Writer Subscribes to UPnP Devices . . . 156
10.2 UPnP OWL Writer Service Generation . . . 157
10.3 SMOA++ search for substrings in WordNet . . . 160
10.4 Second Round Antonyms Search . . . 161
10.5 Part of the Matching Concept Implementation . . . 163
10.6 The Adaptation API Interface . . . 165
10.7 External Service API . . . 167
10.8 Part of the DOXEN Configuration file . . . 168
10.9 Preparing the Activator Information for Template Filling . . . 169
10.10 Part of the Activator Template . . . 169
10.11 Part of the UPnP Proxy Device Generation . . . 171
10.12 Compile Generated Code . . . 172
B.1 OWL Description [Pierson 09] of the Ontology in Figure 5.3 . . . 209
B.2 A UPnP BinrayLight Generated Ontology . . . 212
B.3 An Alignment Result between a UPnP and DPWS Lights in the Align format . . . 217

Terminology

ACS ..... **Auto Configuration Server**
ATOPAI ..... **Alignment and annotation Tool framewOrk for Plug and plAy Interoperability**
BD ..... **Base Driver**
BMS ..... **Basic Management Service**
CMS ..... **Configuration Management Service**
CP ..... **Control Point**
CPE ..... **Customer Premises Equipment**
CWMP ..... **CPE WAN Management Protocol**
DCP ..... **Device Control Protocols**
DLNA ..... **Digital Living Network Alliance**
DOXEN ..... **Dynamic Ontology-based proXy gENerator**
DPWS ..... **Device Profile for Web Services**
GENA ..... **General Event Notification Architecture**
IGRS ..... **Intelligent Grouping and Resource Sharing**
KR ..... **Knowledge Representation**
NAS ..... **Network Area Storage (Device)**
OASIS ..... **Advancing Open Standards for the Information Society**
OSGi ..... **Open Service Gateway initiative**
OWL ..... **Ontology Web Language**
OWL-S ..... **Semantic Markup for Web Services (OWL)**
PnP ..... **Plug and Play**
SMS ..... **Software Management Service**
SOA ..... **Service Oriented Architecture**
SOAP ..... **Simple Object Access Protocol**
SSDP ..... **Simple Service Discovery protocol**
STB ..... **Set Top Box**
UDA ..... **UPnP Device Architecture**
UPnP ..... **Universal Plug and Play**
UPnP-DM ..... **UPnP-Device Management**
XSD ..... **XML Schema Definition Language**
WS ..... **Web Services**
WSD ..... **Web Services on Devices**
WSDL ..... **Web Service Description Language**
WS-MAN ..... **Web Service Management**

Chapter 1

Introduction

Smart homes are becoming a reality with the development of embedded systems, service oriented architecture and "Plug and Play" protocols. These new habitats, where computer systems are anywhere hidden in the environment, fit into the well known ubiquitous systems imagined by Mark Weiser in [Weiser 91].

Devices supporting specific "Plug and Play" networking protocols are capable of announcing their type and capabilities in the home network. Consequently, smart applications and other devices are able to discover and cooperate with such devices to accomplish specific tasks. Such applications can be deployed on smart phones and PCs enabling the interaction between devices (lights, TV, printer) and their correspondent hosted services (Switch Light, Control TV, Print). For example, a Photo-Share application automatically detects an IP digital camera device and on the user command, photos are rendered on the TV and those selected can be printed out on the living room printer. The applications and devices configuration is completely transparent to the user who deploys the application on his home gateway, for example.

Furthermore, the actual device plethora allows a proliferation in the applications development to provide a wide set of services like the multimedia user experience, energy saving, monitoring, maintenance or even surveillance and intrusion detection. Such applications create a new market perspectives which can be shared between three major actors: telecoms operators providing Internet and telephony access through already deployed set-top-boxes and home gateways, manufacturers providing plug and play devices and third parties developing the applications and offering services based on the user home devices through the telecoms infrastructure.

However, the digital home is more than ever a heterogeneous and complex environment. Indeed, devices differ in terms of resources, communication mediums and networking protocols. Thus, such heterogeneity prevents the cooperation and interaction between the devices and applications to fulfill the user requirements and tasks.

Thus, we focus in this work our effort to provide an interoperability between devices and applications supporting different protocols. We depict in the next section, the heterogeneity problem retaining the interaction between the smart applications and the "Plug and Play" devices.

1.1 Problem Statement

Devices supporting a *"Plug and Play"* protocol publish their capabilities and type on the network. More precisely, a device is constituted of three entities. The first is the protocols stacks used to communicate and interact with other devices and applications. The second entity consists in the provided services which represent the devices' capabilities. And finally, the third entity is the device and service description which are announced on the network. Based on the announcements, the applications deployed in the home network discover the provided services and interact with devices and applications to accomplish a specific task, such as dimming the

light intensity or printing a document.

Currently, the following four plug and play protocols are widespread: UPnP, DPWS, IGRS and Apple/Bonjour. Such protocols cohabit in home networks and share a lot of common features. They all announce their descriptions and can be remotely controlled by other devices and applications on the network. They also target similar device types. Multimedia devices are shared between UPnP, Bonjour and IGRS while the printing domain (printing, scanning) is dominated by UPnP, Bonjour and DPWS. More over, each protocol defines standard profiles with required and optional implementations that manufacturers need to support.

Even though those protocols have a lot in common, applications and devices can not cooperate due to the three following differences:

- **Protocols Stacks**: each protocol defines its own protocol stacks which is based on several protocols to announce the device description and to support remote interaction on the network. Thus, applications not supporting the same protocol stacks cannot discover the devices announcements and interact with.
- **Description Format**: each plug and play protocol proposes its own device annunciation and description format. Thus, the applications must be able to interpret the announced description on the network in order to discover the device type and capabilities.
- **Description Content**: each protocol provides a standard content description per device type. The content description includes the device type name, the required supported services names and versions. The standard content description also defines the names of the supported operations on the device along with their input/output parameters names, types and ranges. Thus, two standard equivalent devices such as the printers, supporting different plug and play protocols announce a syntactically different but semantically equivalent content. For instance, a UPnP light hosts a `SwitchPower` service with a `Switch(true/false)` method to control the light while a DPWS light[SOA4D b] uses the semantically equivalent method `SetTarget (On/OFF)`.

Thus the previously outlined three levels of heterogeneity: the description content, format and the protocols layers diversity, prevent applications to use any available equivalent device on the network to accomplish a specific task, such as printing a document or rendering a song. Designing applications to support multiple protocols is time consuming since developers must implement the interaction with each device profile and its own data description. Additionally, the deployed application must use multiple protocols stacks to interact with the equivalent devices. We believe that home applications should be set free from such heterogeneity and should be able to interact with any device capable of fulfilling a desired task. More over, application vendors and telecoms operators need to manage and orchestrate devices through a common application layer independently from the protocols layers and the devices descriptions heterogeneity.

1.2 Contributions

To tackle the devices heterogeneity and to accomplish interoperability between plug-n-play devices and applications, we propose to represent non-UPnP devices as standard UPnP devices by automatically generating proxies. Each proxy announces itself as a UPnP standard device on the network and controls an equivalent non-UPnP device. Thus, UPnP applications can interact with non-UPnP devices transparently through the UPnP generated proxy. The use of the UPnP profile description and stack as a common pivot is due to its wide acceptance among device manufacturers and vendors. However, another protocol pivot and profile can be chosen instead, since our approach is generic and is independent from a specific protocol.

The contributions of this thesis focus on providing to the applications targeting specific devices, with specific protocols, the ability to interact with any equivalent device on the network. In other words, our approach

provides semantic and behavior interoperability between two equivalent devices based on the intersection of two major domains. The ontology matching crossed with the model driven engineering domain. The main contributions of this thesis are three fold. The first two contributions propose a scientific solution to the heterogeneity problem. The last fold is a technical contribution which proves the realization of the first and second contributions.

- **An End To End Adaptation Solution:** To allow a cross protocol interaction between UPnP applications and non-UPnP devices. We propose in this thesis an end to end novel solution to provide Plug and Play interoperability. The solution, consists of six major steps [El Kaed 10, El Kaed 11a], as shown in Figure 1.1. The first step, the *"Ontology Generation"*, resolves the description format heterogeneity by automatically generating ontologies representing the devices descriptions in a uniform ontology format. The ontology expresses the devices capabilities independently from the technical details and the specific protocol description format.

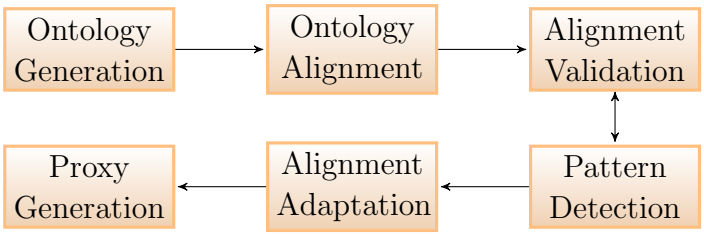


Figure 1.1: Steps of the approach

The second step, the *"Ontology Alignment"*, attempts to resolve the content description heterogeneity. It applies on the previously generated ontologies semi-automatically matching techniques. The matching aims to detect correspondences between two equivalent ontologies which represent two devices having the same type. The ontology alignment detects the translation rules to go from one device capabilities to another.

Since the matching techniques are heuristics based, the third step, the *"Alignment Validation"*, requires a human intervention to validate and edit the detected correspondences. Based on the validated alignments, the fourth step, the *"Pattern Detection"*, applies rules to detect services and actions compositions based on their parameters. The rules automatically annotate the ontology when a composition pattern is detected. This step also detects correct compositions and returns to the expert non valid ones. The expert, based on the devices specifications and behavior, tries to adapt the non valid compositions by adding an adaptation behavior.

The fifth step, the *"Alignment Adaptation"*, is an optional step. It depends on the previously detected correspondences and compositions. This step allows the expert to add an adaptation behavior using a high level language.

The final step, the *"Proxy Generation"* consists in exploiting the validated correspondences between the two devices descriptions to automatically generate proxies [El Kaed 11b]. The generated proxy acts according to the validated correspondences which contains the adaptation behavior to allow a transparent interoperability between the two devices capabilities. For instance, a UPnP application can now interact with an available DPWS Printer as a UPnP standard Printer. In fact, the generated proxy exposes itself as a UPnP printer and adapts the received invocation from the UPnP application to invoke the DPWS printer.

The protocols stacks heterogeneity is resolved in this final step by the generated proxy which relies on

a technical solution. The generated proxy uses a high level API to abstract the technical details of the protocols stacks.

- **Matching Technique**: The second contribution of this thesis occurs in the second step, the *"Ontology Alignment"*, of the solution. We propose an ontology matching technique inspired from the SMOA [Stoilos 05] matching approach. Our technique relies on a semantic dictionary which allows to detect synonyms and antonyms relations between two entities from the two ontologies.

- **Prototype Validation On Real Devices:** In order to prove the feasibility of our proposed solution, we implemented a prototype of three modules. The first module handles the automatic ontology generation from devices annunciations. The second module allows to trigger the ontology matching and to validate the alignments. And finally, the third module automatically generates proxies based on the previously validated ontology alignments. The automated generation, provides an adaptation without any human intervention in the proxy implementation process, which is time consuming and error prone.

  We tested our approach on three different devices: lights, clocks and printers. The evaluation proved that our approach is applicable on simple devices such as the lights and the clocks. More importantly, the evaluation shows that our approach is scalable and can be carried out on complex devices like the UPnP and DPWS standard printers. In fact, the two printers are the two most complex standardized devices so far proposed in the UPnP and DPWS forums. Furthermore, today, it seems that we are the first to provide the interoperability between the two standard printers.

1.3 Thesis Outline

We expose in this section the organization of this document which details our proposed approach to resolve the devices heterogeneity. This document is organized in three parts.

- The first part outlines the context of our work as follows:

  Chapter 2 depicts the ubiquitous computing environment through a scenario, then extracts the ubiquitous computing characteristics and challenges. Based on such challenges, we propose the characteristics of an ideal ubiquitous system which are mainly the following: control, eventing, discovery and interoperability. Then, we overview the actual state of the digital home environment which tends towards a ubiquitous one. In **chapter 3**, we overview the service layer of a device in a service oriented architecture (SOA) vision. Then, we insist on the SOA characteristics and show how they meet the ubiquitous system characteristics. And finally, the chapter outlines an SOA based framework, *"OSGi"*, along with its architecture. The SOA supports the characteristics of a ubiquitous system, however, the interoperability between the devices remains unsolved. Therefore, in **chapter 4**, we first detail the following play protocols: UPnP, DPWS, IGRS, Bonjour along with their underlying stacks. Then, we draw a comparison between such protocols and detail their common and divergence points. Finally, we highlight the three layers retaining a complete interoperability between the plug and play devices: protocols stacks, description format and content heterogeneity. Since, the protocols stacks heterogeneity can be tackled by an OSGi based technical solution, the description format and content heterogeneity remain. Thus, in **chapter 5**, we point out the knowledge representation in general then, we go through the service and device description languages and formats. We overview three knowledge representation techniques which can be useful to resolve such heterogeneity. **Chapter 6** concludes the part and details the problem statement.

- The second part overviews the related work.

Chapter 7 first outlines the currently existing solutions providing interoperability between heterogeneous services and devices. Then, the chapter classifies the proposed approaches into three categories. A detailed comparison is presented to expose the advantages and drawbacks of each category. Based on the comparison between the proposed approach, **Chapter 8** gives an insight on the advantages of the ontology matching techniques to resolve the description content layer.

- The third part presents our contribution. **Chapter 9** details the contribution of this thesis. A first section presents the motivation and the overview of the approach. Then, the automatically ontology generation is exposed. Another section introduces the device matching, the ontology alignment techniques and strategies. Then, we detail the ontology validation performed by an expert. The rule-based pattern detection to automatically annotate the ontology and detect services' methods compositions. Then, we detail the global overview of the architecture with regard to the digital home's and the operator's sites. And finally, we overview the proxy generation based on the ontology alignments.

  Chapter 10 outlines the implementation of our approach and exposes the carried out experimentations on several devices. **Chapter 11** evaluates each module of our proposed approach. The first section evaluates the ontology generation while the second section details the device matching and ontology alignment evaluation on three equivalent devices types. The last section reports the evaluation results of the proxy generation module. **Chapter 12** concludes the thesis and presents the major perspectives of this work.

Part I

Context

Chapter 2

Ubiquitous Computing

"Ubiquitous computing names the third wave in computing, just now beginning. First were mainframes, each shared by lots of people. Now we are in the personal computing era, person and machine staring uneasily at each other across the desktop. Next comes ubiquitous computing, or the age of calm technology, when technology recedes into the background of our lives."

– Mark Weiser

Contents

2.1 Ubiquitous Scenario **27**

2.2 Ubiquitous Environment Characteristics **32**

2.3 Ubiquitous System Characteristics and Challenges **35**

2.4 The Digital Home Towards a Ubiquitous Environment **38**

2.5 Conclusion **44**

In this chapter, we first introduce the context of ubiquitous environments through a well detailed scenario covering various aspects of such an environment. Then, in section 2.2, we extract from the previously detailed scenario, the ubiquitous environment characteristics. From such environment characteristics and features, we point out in section 2.3 the characteristics and challenges that a ubiquitous system needs to handle. In section 2.4, we detail the current state of the digital home and the challenges to be faced. We also provide an overall layer architecture of the applications and devices of the digital home. Then, we discuss the scope of each characteristic with respect to the actual device and application architecture. Finally, we present a conclusion of this chapter and outline the challenges to resolve in our work.

2.1 Ubiquitous Scenario

Mark Weiser defines in [Weiser 91] the ubiquitous computing as an environment with inter-connected computers, used by humans unconsciously to accomplish everyday tasks. In such environments, a computer is a machine capable of one or more operations like analyzing, processing, storing and transmitting information. Nowadays and thanks to the development of embedded technologies in the twentieth century, computers are embedded in thin devices having different memory and CPU resources ranging from scare resource devices like sensors measuring ambient temperatures to more abundant resource devices like laptops, set-top-boxes capable of processing high definition videos.

Three essential elements constitutes the ubiquitous computing:

- Devices ranging from tiny sensors measuring temperature, to home appliances machines such as heaters or consumer electronics like digital cameras and televisions. Such devices provide one or more services along with their control commands. For example, a light device hosts a `SwitchPower` service and the action `Switch` to turn on or off a light, a printer offers a `Printing` service along with actions like `Print` to print a document or to cancel it with `CancelPrint` action.

- Ubiquitous applications acting as orchestrators by controlling devices to accomplish a specific task. The aim of each application is to hide the devices interaction complexity and permit users to manipulate a large set of devices unconsciously to accomplish their needs. For example, on a ring-door button notification, an application activates the camera of the front door and redirects the video stream to an adequate display interface near the user so he can identify the visitor.

- A network inter-connecting the different devices and transporting the events and notifications to the correspondent application or device. The network is an assembly of different network types ranging from short range communication like *Bluetooth* [Bluetooth] and infrared to medium range communication like the *Wifi* or the Ethernet or even a wider area like the cellular network and the Internet.

Figure 2.1: Ubiquitous Environments

Different applications have been proposed in the ubiquitous environment. We detail next a ubiquitous scenario providing several examples of ubiquitous applications. Then, we extract from the scenario, the environment the arising characteristics and challenges.

In the left part of figure 2.1, "Bob" installed a ubiquitous application in his home which orchestrates home devices upon human activities such as the lights, the blinds and the coffee machine. As soon as "Bob" lies on his bed at night, sensors disseminated in his bed notify the home application.Based on his current movements and his previous sleeping behavior [Helaoui 11], the application is capable of inferring if "Bob" is currently sleeping. Then, the application ensures that no lights are involuntary forgotten on and turns them off. It also activates the security monitoring system and alarm.

Eventing is an essential feature in the ubiquitous environment, devices disseminated in the environment like sensors notify applications instantly when an event occurs

Querying Contextual Memory to predict a certain behavior. The contextual memory is a sort of collected data from different sources in the environment. It can be set by the user or calculated over a period of time. Applications access such information to infer behaviors like Bob's usual sleeping hours.

In the morning, the alarm notifies the application that it is time to wake up "Bob", therefore, the ubiquitous application opens the bedroom blinds, activates the bedroom speakers, plays his preferred music from his smart phone and starts the coffee machine.

When "Bob" picks up his cup of coffee and heads to take his breakfast, the application fades out the music and turns on the kitchen Television to display the daily news from his preferred channel along with the current weather forecast of Grenoble city where he lives. Once "Bob" finishes his breakfast, the home application requests from the fridge and the kitchen cabinets an update of the food missing. Once received, the application notifies him that there no milk and cereals left for the next breakfast. Thus a grocery list is displayed on the kitchen Television so he can add or/and remove items. "Bob" validates the list and chooses to pick up the grocery on his way back from work, the home application sends the command to the grocery shop.

Interoperability between ubiquitous applications is primary to communicate with another environment to propose more services. The "Home Application" interacts with a weather forecast application and renders it on the TV. It also interacts with a"Grocery Application" to command missing goods.

As soon as "Bob" leaves the home with his smart phone and laptop, the application closes the blinds, turns off the lights and re-activates the security monitoring system. When "Bob" is identified by his car application, it opens the door and requests from the home application the last channel visited by "Bob" recently, then activates the car radio and sets the channel. The content of the smart phone and the laptop become available in the car environment. On the way to work, the car applications connects to a global positioning system (GPS) and guides him through the streets where there is less traffic. The car application is also notified that "Bob" has to pick up the grocery on his way back home.

Once arrived at work, the car application requests from the "work application" available parking spots and guides "Bob" to the nearest spot. Then, the car application notifies the work application that "Bob" has arrived and he is entering the office. Thus, the work application renders on his office display screen (laptop or PC wide screen) his important meeting hours of the day along with the high priority received emails.

Mobility, personal devices follow the user from one ubiquitous environment to another.

Device and User location enable ubiquitous applications to migrate services to the new location and therefore offering a service continuity. The car application tuned the car radio to latest channel previously set by "Bob" at home.

Communication mediums between devices and applications are heterogeneous, the car uses a GPS, a laptop uses a Wifi network while a smart phone uses a 3G network.

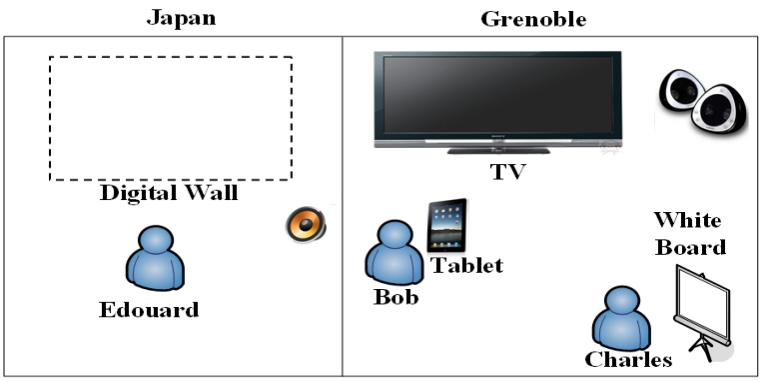


Figure 2.2: Ubiquitous Work Environments

At 3 PM, *Bob* starts a scheduled meeting with his coworkers, *Charles* present in *Bob*'s office and *Edouard* in Japan. The work applications on both sites, mute all the four smart phones and other notifications since *Bob* and his colleagues agreed that they should not be bothered during the meeting. The work applications on both

sites (see Figure 2.2), activate the disseminated cameras and microphones along with the speakers and close the blinds.

Bob takes his tablet and starts the meeting application, a list of available display devices in his office is shown. He chooses the wide screen television in his office as the main display screen while *Edouard*'s chooses the digital wall of the meeting room. *Bob* starts the meeting by overviewing the headlines of the cooperation terms with the Asian firm. *Charles* takes the presentation role and proceeds with the meeting, he approaches a white board and would like to draw a schema to represent the product they will be working on. *Charles* prefers to draw on real papers instead of digital ones. With a gesture, *Charles* notifies the disseminated cameras to focus on the white board to show *Edouard* what he is currently drawing. The meeting application is notified, it activates the video projector device. The work applications connect the digital wall and the wide screen television in both sites. They are now capable of rendering the white board drawing. *Edouard* updates the schema by drawing on his digital wall, the work applications synchronize the content and update it on the display screens, the video projector also displays a projected image update of *Edouard*'s drawing on the white board. *Bob* would like to clarify an aspect of the drawing, therefore, he draws directly on his tablet. The work applications update *Bob* clarification on the display screens. Two hours later, the meeting comes to its end, the work applications display the work calender of each participant along with the common available time for another meeting. The applications then update each participant work calender with the schedule of the next meeting and notifies *Bob* that he will be in Asia for the next meeting.

Gestures Interpretation is a new human computer interface, an application recording the meeting detects a human special gesture to command a device or integrate a physical object into the digital environment.

Integrating physical objects into ubiquitous environments using cameras and video projectors for example allows to transform static objects like white boards into a digital support to share information with other users.

On his way back home, *Bob* would like to have a Pizza for dinner, therefore using his smart phone, he sends his order to his preferred Pizzeria. The smart phone notifies the car application that a Pizza order has been issued and the home itinerary need to be adjusted to pick up the Pizza and the grocery on the way back. The car application recalculates the fastest itinerary with the minimum traffic.

Once at home, as shown in the central part of figure 2.1, *Bob* enters the living room and sits on his chair in front of his wide screen television. *Bob* browses an online movie repository and chooses one. Upon the movie selection, the home application is notified and since the living room chair already notified *Bob* presence. the application is aware that Bob is in the living room, therefore the living room TV will be used to display the movie. Thus, the home application sends commands for multiple devices in the living room, the blinds are closed and the lights are dimmed, the Hi-Fi system is activated and is ready to receive the TV audio output.

After a while, the home application receives a notification from the smart phone placed on the entrance hall battery charger that *Alice* is visiting *Bob* in an hour and he did not yet acknowledge the notification. The application redirects this important notification to the wide screen television in order to attract his attention along with the remaining time of the movie. *Bob* acknowledges the notification and continues to watch his movie.

Device Interoperability allows an efficient device and service orchestration to accomplish a specific user task, TV, microphones and speakers are orchestrated to enable a meeting session or a home cinema experience.

An hour later, *Alice* arrives with her digital camera, she just came back from her safari trip and wants to share her photos of animals and wildlife with *Bob*. In the right part of figure 2.1, the digital camera identifies itself and announce its supported services and capabilities, thus, the home application notifies *Bob* that a new device has been detected and asks if it should be integrated to the home network. After its integration, *Bob* uses his smart tablet to display *Alice*'s photos on the wide screen television and to flick her safari album. While seeing the photos, *Bob* stores his preferred ones on his living room Network Attached Storage (NAS) device. He also prints out some of her photos on his living room printer.

Device Discovery and Interaction are necessary to integrate new devices into the ubiquitous environment. The discovery includes identifying the device type and its supported services.

Alice also videotaped great scenes of animals that she stored on her living room NAS. To watch the videos, *Bob*'s home application interacts remotely with *Alice*'s home application, and after security checks, the movies can be remotely displayed on *Bob*'s television.

Security and Privacy are naturally essential to prevent malicious access to personal devices containing sensible and private information.

Roles must also be defined to specify the access and allowed operations of each user and application. For example, Alice's roommate has access to the NAS and can only view some of Alice's photos. No delete operations are allowed.

After a while, *Alice*'s smart phone notifies her that she has a wine tasting activity in an hour in a near by restaurant. *Alice* asks *Bob* to join her. Once arrived, the restaurant application redirects the wine menu to *Alice* and *Bob* smart phones. The menu also details information about each wine type such as the year of harvest, the region, the climate and the wine characteristics. *Alice* and *Bob* can order the wine directly or ask for a waiter assistant to help them choose.

The next day *Bob* has to travel to Asia early in the morning for a week to complete the cooperation agreement with the Asian firm in *Japan*. Therefore, the home application adapts his home alarm to his work schedule. The home application checks his destination and downloads on his smart phone the Japanese to French audio/visual translator along with Tokyo's city map, weather forecast and his hotel reservation. The application noticed a six hours free time between his meetings and his way back flight. Therefore, the application proposes a touristic city tour with the major monuments to visit in *Tokyo*.

Inference allows to predict certain needs for the user, such as predicting a dictionary or a map download based on his destination trip.

Once arrived at the airport, his smart phone notifies the check-in application that *Bob* has arrived and requests the check-in gate for his flight along with the gate and seat information. The smart phone displays these information and guides *Bob* in the airport. After his landing in *Tokyo*'s airport, his smart phone traces the itinerary to follow from the airport to the hotel where *Bob* is staying. He takes the subways and on his way, *Bob* approaches his smart phone from the advertisement bill board and activates the visual French-Japanese translator which redisplays the advertisement in French. *Bob* arrives at the hotel, his smart phone connects to the "Hotel" application and notifies that *Bob* has arrived and checks in. Then the smart phone displays *Bob*'s room number and floor.

During his stay, *Bob* used his smart phone for multiple purposes, for example, the smart phone displays the Japanese restaurant menus in French and allows to order food without any knowledge in the Japanese

language, *Bob* also used the audio Japanese to French translator application which takes an audio as input and translates it to French to understand the Taxi driver and able to communicate with. During the city site seeing tour, *Bob* used his smart phone to move around the city and to obtain more information about a monument. For example, he uses a monument recognition application, *Bob* only places his smart phone in front of the monument, then the application identifies it and connects to the Japanese ministry of tourism or *wikipedia* for example to download more information about it. The smart phone also notifies *Bob* that it is time to head to the airport to take his flight back to *Grenoble*.

Content Adaptation is a content transformation which includes language translation, an audio to text display transformation and an appropriate display adaptation of information in conformance to the display device type. Display information on a TV is not the same as on a mobile or a digital bill board.

Offices, homes, airports, restaurants and even more are an excellent example of an ubiquitous environment where devices are interconnected through a network and orchestrated by applications to accomplish a task. In the following section, we extract from the previously detailed scenario some characteristics of the ubiquitous environment and then detail the arising characteristics of a ubiquitous computing system.

2.2 Ubiquitous Environment Characteristics

In the light of the previous detailed scenario, we extract and analyze in the following section the major characteristics of the ubiquitous environments.

2.2.1 Dynamicity

A ubiquitous environment is a highly dynamic one, where devices join and leave the network due to several reasons, we summarize in the following some of such reasons:

User Mobility Personal devices dynamicity is correlated with the end-user mobility. Smart phones, digital cameras and laptops usually follow end-users in space and time from one ubiquitous environment to another. Applications are expected to handle the service departure proposed by those devices. For example, when a digital camera leaves a ubiquitous environment, the application marks its functionality as unavailable. Such personal devices are also expected to be integrated transparently into the new environment along with a smooth interaction with existing devices in the new environment.

Device Energy The development of embedded devices as predicted by Moore's Law [Moore 65] where processing capacity doubles approximately every two years allows such devices to handle more tasks and therefore consume more energy. The display screens and wireless cards are also greedy in energy despite advances in battery manufacturing and energy saving techniques. An average energy autonomy of a laptop holds between 3 to 9 hours, therefore, the device availability is dependent on its energy. Meaning the dynamicity is dependent on the device energy.

Device Non Usability Devices are also disconnected when there is no need for its functionalities at some given time, for example, the heating system is turned off during the summer season.

Replacing Devices The end-user is also expected to renew his devices, i.e. replacing a device with another or adding a new device to his home.

Device Proteanism Devices have become recently multi-functions supporting different roles and functionalities. For instance, a smart phone is used to make phone calls, take pictures and videos, listen to music and even as a remote controller. Therefore, the user might switch off a functionality and activate another one on the same device. The environment will perceive a new capability arriving to the network and another leaving.

The high dynamic nature of ubiquitous environments requires highly dynamic ubiquitous applications able to detect the device arrival and departure. Upon a device arrival, the application need to detect or to be notified of its capabilities to enable and apply the orchestration among different devices and applications.

2.2.2 Heterogeneity

One of the most challenging features in ubiquitous environments is heterogeneity which is present on different levels, we outline some of them in the following:

Resources A wide variety of devices co-habit in ubiquitous environments and ranges from scare resource devices like sensors disseminated in the furniture detecting movement to a more abundant resource devices like laptops, set-top-boxes capable of processing high definition videos. This resource heterogeneity translates into a difference capabilities of storage, computing and communication. A temperature sensor has less computation power and transmission range then a Set-Top-Box capable of treating data streams and connecting to other devices in a wireless local area network.

Ubiquitous applications are installed on devices in ubiquitous environments and share the common resources of their underlying physical devices. Therefore, the resource heterogeneity must be taken into account to allow a fluent ubiquitous computing experience for the end-user.

Communication Mediums The difference in resources imposes the use of different communication mediums, an embedded sensor uses low power radio communications such as infrared or low power radio waves mediums to exchange low amounts of data with other sensors or devices. Lights and Switches can use *PLC* (Power Line Communications) to exchange data over electrical power wires [Yousuf 07]. A smart phone uses a mobile telecommunication medium like the 3G or UMTS and can also switch to a wireless local area network to communicate with other devices like printers or TVs. Ethernet is also used to connect devices with relatively abundant resources like Set-Top-Boxes and TVs to exchange video/audio streams. A car uses the GPS to communicate with satellites using special radio waves needing more computational power and energy. A network is then generally constituted of devices communicating through the same medium. Gateways are used to connect different networks and devices.

The communication medium diversity forces ubiquitous applications to support different medium communications or use a multitude of gateways in order to interact with multiple devices supporting heterogeneous communication means.

Communication Protocols General purpose protocols are being proposed on top of the communication mediums, which can be classified into two categories, the IP based protocols and non-IP based protocols. The IP-based offer a lot of benefits such as packet routing, communication reliability, packets ordering, etc. Therefore, the IP-based protocols require devices with enough computational and transmission capacities. However, the non-IP based protocols like *ZigBee*[ZigBee 09] or *Bluetooth*[Bluetooth] are deployed on low power and scare resource devices to exchange small amounts of data without the need to handle packet routing or reliable communications. The data packets size is smaller than the IP based protocols packets size. Therefore, no extra communication and computational capabilities are needed. For example, the IETF proposed recently the 6LowPAN[IETF 07] (IPv6 over Low power WPAN) for low power communication.

On top of these protocols layers, specific application protocols layers are added for a specific purpose like the *DHCP* [Droms 97] (Dynamic Host Configuration Protocol) allowing devices to acquire an IP address. Other protocols have been used by devices to announce their services descriptions along with their capabilities like the *SSDP* (Simple Service Discovery protocol) [Goland 99] or the *Web Services Discovery* [OASIS 09b]. The *GENA* (General Event Notification Architecture) protocol or the Web Services Eventing [W3C 06b] are also employed by devices for notification and event broadcasting.

Other protocols built on top of these previous protocols layers are proposed and implemented by devices to communicate in ubiquitous environments, like UPnP the Universal Plug and Play Protocol [UPnP], DPWS The Device Profile for Web Services protocol, IGRS the Intelligent Grouping and Resource Sharing (IGRS) [IGRS] or the Apple Bonjour [Bonjour] protocol.

However, ubiquitous applications are expected to communicate transparently with devices regardless of the protocols they use.

Software and Hardware platforms The heterogeneity of resources, communication mediums and protocols, constitutes major factors influencing the hardware device components selection and constitution. The device resources and hardware, influence the software platforms selection. Such software platforms varies from tiny and embedded operating systems deployed on sensors to operating systems and virtual machines deployed on servers and devices with abundant resources. Thus, the hardware and software platforms are also heterogeneous and vary from a device to another. Moreover, ubiquitous applications are defined as software components running on software platforms to accomplish users tasks. Such components cannot be installed on any platform since they depend and rely on the functionalities offered by the specific underlying software and hardware platform. The diversity of the software platforms makes installing ubiquitous applications on any available device with enough resources and capabilities a difficult task.

Device and Service Description Devices arriving to a ubiquitous environment are expected to identify themselves along with their supported services and capabilities. However, a lot of XML-based description formats are being used to expose the device description and capabilities. For instance, devices supporting the UPnP protocol uses an XML based format while other devices supporting the DPWS protocol uses the *WSDL* (Web Service Description Language) to expose the service descriptions.

Other than the description format, the syntax content is not the same even for the same device type and functionalities, for instance a UPnP Intel Light uses the "`BinaryLight`" to identify the device type while a DPWS Light declares its type as "`SimpleLight`". The services and the actions have also different syntax, the UPnP Light offers a "`Switch`" service with a `"SetTarget"` action to turn on or off the light, while the DPWS light uses the action `"Switch"` to offer a similar functionality.

Ubiquitous Applications deployed in ubiquitous environments are expected to identify devices along with their services and integrate them in the ubiquitous environment so they can be used transparently by users, other devices or applications.

User Interfaces The classic mouse and keyboard are not the only input interfaces, touch screens and gestures among others have emerged in ubiquitous environments due to their simple usage. Some users also tend to use statical objects like the old white board to interact with other users or with the ubiquitous applications. Therefore special gestures are used as a human interface to interact with the applications has become very promising [Sharma 98].

The output display interfaces also became diversified with advances in display screens and video projectors technologies. The display interfaces ranges from flexible paper-thin screens to smart phones and wide TVs and

walls. Ubiquitous applications are expected to use any display interface near to the user to deliver information. For instance, an unacknowledged meeting on the smart phone will be redirected to the main display screen in the room to attract the user attention so he can validate the meeting.

2.3 Ubiquitous System Characteristics and Challenges

Different challenges arise from the previously enumerated ubiquitous environment characteristics. A ubiquitous system must handle such challenges. Such system is defined as a centralized or distributed middleware deployed on different devices in one or multiple ubiquitous environments. The middleware executes software components which represent among other service, the ubiquitous applications. An application offers or consumes services provided by other applications or devices. For instance, the grocery shop application offers the ordering items service to other applications. The grocery application also consumes a secure on-line payment service provided by another application. Devices also offer or consume services. For instance, the "*Bob*"'s Home application interacts with services provided by home devices, like the "Switch" service provided by a light to allow turning on or off the light.

We detail in the following the characteristics and challenges of an ideal ubiquitous computing system.

The Ubiquitous computing vision consists in helping users to accomplish their everyday tasks by easily using devices and technology. To achieve such target, ubiquitous applications must be aware of the environment surrounding the user in order to propose services that fit at best his needs. For instance, an application aware that the user is sleeping will mute his smart phone and turn off all forgotten lights in his home.

The user mobility and the device dynamicity are the main factors leading ubiquitous applications toward **context awareness** in order to fulfill the users need. To achieve such awareness, applications receive information from different resources: the user, other ubiquitous applications or devices. The user interacts with the environment, he actually activates the ubiquitous applications, replaces devices, triggers notifications and informs applications about his preferences. For example the user sets his favorite news or music channel. Devices like smart phones, lights or motion detectors disseminated in the environment communicate their notifications to the applications for multiple reasons, such as informing about an accomplished task like a printed document or to raise an alarm upon fire or smoke detection or simply to announce its services description and capabilities. Receiving information from the context and the surrounding environment implies that an event took place, it is then up to the ubiquitous application to decide whether to ignore it or to react. A representation of an ideal ubiquitous system is exposed in Figure 2.3 where different layers are shown. The first layer (L1) contains devices in the environment capable of sending notifications and information to the application. The static objects can also be integrated in the ubiquitous system using devices such as cameras and video projectors. In the previous scenario, *Charles*'s white board is integrated in the system through disseminated digital cameras capable of detecting human gestures. The (L2) context aware layer in Figure 2.3 gathers notifications from various devices and applications then notifies the applications that an event occurred. Then it is up to the application to ignore then event or to treat it. The reaction takes different aspects: an ***adaptation*** of the environment, an ***auto adaptation*** of the application or both. The adaptation can be handled by the system, the application or both.

Adapting the surrounding context occurs by interacting with devices and other applications. For instance, an application opens the blinds in a room to adapt the environment and restore a predefined level of light intensity.

Another form of adaptation is the auto adaptation which is the ability of an application to extend or remove features depending on the context change. For example on a device discovery, the system handles the device arrival and departure then informs the applications which adapt and propose an extended feature allowing users to accomplish more tasks such as the possibility to print a document on the newly discovered device. The

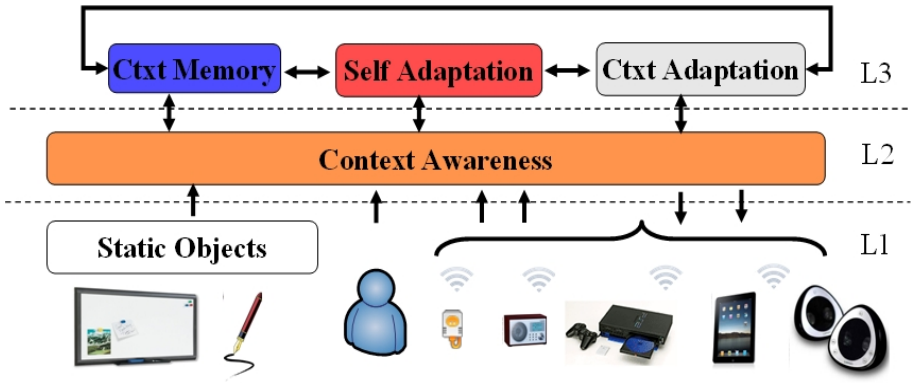


Figure 2.3: Ideal Ubiquitous System Architecture

application should also remove certain features related to non available devices on the network. For example, on a smart phone disappearance, a "Media share" ubiquitous application should be aware of such departure and expose the media content of the disconnected device as not available at the given time.

More often, adaptation includes the environment and the self adaptation. The content adaptation is an excellent example of both adaptations. For instance, a displayed text on a smart phone screen will not be the same on a TV or on a giant billboard. A redirected notification or information firstly received on the smart phone will be reshaped and enlarged when redirected on a wide screen display.

Moreover, ubiquitous applications should be able to use and access some sort of a ***"Contextual Memory"*** containing previously entered users or applications preferences and actions. Therefore, applications request the user preferences and habits in order to adapt the context. For example, by setting *Bob*'s favorite channel or even infer and predict assumptions based on previously collected information. For example, since no movement action has been detected in the house and since *Bob* slept at midnight all this month, the application can infer that he is currently sleeping, thus, the application can turn off his house lights.

Additionally, ubiquitous applications must be capable upon user request of integrating static objects such as a paper, white boards or pens into the digital environment. For example, a ubiquitous application recognizes a user gesture in order to render the white boards content on the television? Thus, the static object becomes a part of the digital ubiquitous environment. User gestures represents the peripheral between the digital and the physical world.

We outline next the characteristics and challenges.

2.3.1 Discovery

The device discovery constitutes a serious challenge in the ubiquitous computing environment since devices are highly dynamic. Applications must be aware of the device arrival and departure in order to apply adaptation on the context or on the application itself. Therefore, a device must first join a network and requests an address identifying it on this network. Then, the device must announce its arrival along with its description which includes, for example, its device type, name, manufacturer, serial number, software and hardware version along with its supported services and capabilities. The description annunciation allows applications to uniquely identify a device along with its capabilities and check if it responds to the requirements in order to interact with it to accomplish a given task. The device should also announce its departure on the network.

Moreover, the device physical location in the environment is an essential information. For example, a user would like to view photos stored on his friend's NAS (Network Attached Storage) the one located in the living room. Applications also need to access the device location, a motion detector notifies that a person has just

entered the room, based on this information an application activates devices around the user's new location to ensure a continuity of a certain service or to accomplish a given task. For example the home application activates *Bob*'s TV located in kitchen as soon as he enters to display the daily news, or dims the living room lights while watching a movie.

Therefore, we differentiate between the device discovery and the service discovery. The device discovery is the ability to identify a device presence on the network along with its physical location. The service discovery is identifying a device provided services and capabilities. Thus, to be discovered a device should perform the following operations:

- Addressing: A device joining the network is capable of acquiring a valid networking address.
- Annunciation: The device announces its description, including its general information (type, manufacturer, etc) along with its supported services and capabilities.

2.3.2 Control

The control consists in interacting with the device for multiple reasons, such as retrieving its information, requesting tasks.

Discovering a device and its capabilities is not enough for an application to control it. The device must also announce, how ubiquitous applications should interact with its services and supported capabilities. Therefore, a device must also describe how to use a service and its functionalities and the input/output parameters. For instance, a light device describes its supported service "`SwitchPower`" along with the supported actions "`Switch`" to turn on or off the light and the "`GetStatus`" action to retrieve the light status. The description details also that the action "`Switch`" takes a boolean variable "`true`" or "`false`" as an input and also details that the "`GetStatus`" returns an output boolean value. Such information will allow ubiquitous applications to discover the device capabilities and functionalities. Thus, it will allow applications to perform device control and interaction.

2.3.3 Eventing

Eventing has a high impact on the ubiquitous application reaction and adaptation. It consists in informing other applications and devices that an event has occurred. Such event can be a device arrival or departure, an accomplished task or parameter change. The eventing allows applications to be context aware.

There is different sorts of eventing, the pulling is when a device sends information regularly, upon change or upon task accomplishment. For instance a temperature sensor sends the ambient temperature value every hour, while an intrusion detection sensor sends an event upon an intrusion detection. Eventing can also be upon request, an application subscribe to receive notifications regularly, upon a value change or upon task accomplishment. In this case the device only notifies the subscribed applications, instead of broadcasting non-interesting information on the network to multiple applications and devices.

2.3.4 Interoperability

One of the most challenging characteristics of the ubiquitous environment is the heterogeneity. Devices communicate with a multitude of communication mediums and protocols. Additionally, devices uses multiple description formats to detail their capabilities, their supported services along with their functionalities.

The interoperability is the ability of an application or a device to interact with different devices or applications supporting different protocols and representation description. The ubiquitous system must be able to integrate

any device in the environment and interact with it independently from its underlying supported protocol and description format.

2.3.5 Inference

The inference is the capability of the applications to take a decision of adaptability by correlating previous and present information. Such information can be collected by the system through a period of time, like the sleeping hours or set by the user like the preferred radio station.

2.3.6 Interpretation

The system should also be able capable to interpret human gestures which constitutes a new peripheral. The gestures allows users to easily interact with the system or the ubiquitous applications to accomplish a given task. For instance, a gesture can be used to raise up the volume, or close the blinds or even integrating a physical object into the ubiquitous digital environment. In the scenario, we pointed out the gesture interface, however, speech, vision, and touch also represent another computer human interfaces which need to be considered. Thus, the information returned from the peripherals supporting such new communication interfaces must be interpreted to interact and adapt to the humans' needs.

2.3.7 Security

Ubiquitous applications interacts with other applications and devices in local and remote ubiquitous environments. During the interaction, different information are exchanged ranging from the user location, to his media contents and private data. Therefore, a secure system and communication channels are a necessity to preserve the user's data and private information.

Additionally, applications can access devices in the environment to retrieve and share information like photos, videos and documents. Even though, devices are shared among multiple users, the access to private data should be restricted. Roles should be attributed to users and applications, for instance a group or a category of users cannot access contents, another category can have only a read access while a third category can benefits from all the privileges.

We detail in the next section the main actors of the digital home, then we identify the characteristics of such a complex realm and the challenges that arise from each characteristic to be treated in our work.

2.4 The Digital Home Towards a Ubiquitous Environment

The actual digital home is an excellent ubiquitous environment. Multiple characteristics and challenges from the ubiquitous environments are found in today's digital home. In this section, we detail the actual state of the digital home along with its characteristics and challenges.

The actual home device plethora allows a proliferation in the development of ubiquitous applications providing a wide set of services like multimedia user experience, energy saving, monitoring, maintenance or even surveillance and intrusion detection. Such applications create a new market perspectives which can be shared between three major actors: telecoms operators providing Internet and telephony access through already deployed set-top-boxes and home gateways, manufacturers providing certified plug and play devices and third parties developing ubiquitous applications and offering services based on the user's home devices through the telecoms infrastructure[1].

[1]Telecoms operators can also provide ubiquitous applications. Thus, playing two roles.

However, the digital home is an heterogeneous and complex environment for those three actors and the end-user. Devices differ in terms of resources and networking protocols. A wide variety of devices exists in the home and ranges from scare resource devices like sensors measuring ambient temperatures to more abundant resource devices like laptops, set-top-boxes capable of processing high definition videos. Ubiquitous applications are installed on some home devices like STBs and home gateways which share their physical resources. However, telecoms operators and third party application vendors must guarantee a minimum level of QoS (quality of service) when providing their applications. Therefore, the resource sharing between applications has to be taken into account to provide an excellent user experience.

As for the networking heterogeneity, multiple "*Plug and Play*" protocols have emerged like the Universal Plug and Play (UPnP)[UPnP], the Intelligent Grouping and Resource Sharing (IGRS)[IGRS] and the Device Profile for Web Services (DPWS) [OASIS 09a]. Each protocol defines or uses a specific networking layer and a set of standard profiles per device type (printer, light, clock) with required and optional implementation that manufacturers could support. This protocol diversity, prevent applications to use any available equivalent device in the home to accomplish a specific task. For example, a ubiquitous application searching to interact with a UPnP printer cannot use an available DPWS printer. Designing applications to support multiple protocols is time consuming from the developers perspective since they must implement the interaction with each device. For the telecoms operators, such protocol diversity makes the diagnostic and the troubleshooting a more complex operation and adds a burden costs on their infrastructure and tools to target additional protocols. Additionally, the lack of cross-device interoperability, frustrates the users since they will be restrained to a certain protocol and technology if they want a complete interoperation in their home network [Dixon 10].

Moreover, telecoms operators, device manufactures and third parties application vendors provide a maintenance service along with their proposed applications. Thus, devices and applications of the home network must be monitored to provide sufficient information during diagnostic and troubleshooting operations.

Additionally, with thousands of proposed applications, end-users will find them selves confused when it comes to decide if their devices are compatible with the application they are ready to pay for. Therefore, the proposed applications to each end-user should be based on his devices type and software version.

In a such sophisticated environment, we believe that a middleware is essential for the digital home to simplify the complexity for the actors and the end user. The main characteristics and challenges of the digital home are the following:

2.4.1 Discovery and Adaptation for Dynamicity

The digital home is a dynamic environment where devices join and leave the network due to several reasons: personal devices dynamicity is correlated with the end-user mobility. Smart phones, digital cameras and laptops usually follow end-users in space and time from one ubiquitous environment to another. The dynamicity is also dependent on the device energy (battery down) and the non usability of its functionalities at some given time, for example, the heating system during the summer season. The end-user might also replace a device with another or add a new device to his home.

Multiple challenges arise from this characteristic, the middleware must be able to discover the device appearance then identify it along with its hosted functionalities. The arrival of new devices allows to propose new ubiquitous applications to the end-user or to activate features on some already deployed applications. For example, upon the discovery of a UPnP printer, an application can propose to print contents. The middleware should also detect the device disappearance, ubiquitous applications must be notified of such change and should be able to re-adapt.

2.4.2 Interoperability for Heterogeneous Plug and Play Protocols

UPnP [UPnP], IGRS [IGRS] and DPWS [OASIS 09a], cohabit in the home local area network (LAN) and share a lot of common features. Their protocol layers support: discovery, description, control and eventing. Plug and Play protocols follows the service oriented architecture paradigm, devices can be discovered on the LAN since they announce their description (device id, friendly name, manufacturers etc) along with their supported services and capabilities. For example, a printer hosts a service which allows to carry out printing operations. The description allows applications to discover and identify the device in order to control it. Plug and Play devices also support eventing upon a parameter change or upon accomplished tasks.

Those protocols also target similar device types. Multimedia devices are shared between UPnP and IGRS while the printing domain (printing, scanning) is dominated by UPnP and DPWS. Even though protocols have a lot in common, devices can not cooperate due to two main differences: the protocol layers and the service description. Each Plug-n-Play protocol use its own protocol layers: they all use the SOAP protocol for interaction, UPnP and IGRS use SSDP for discovery and GENA for eventing, while DPWS uses a set of standard web services protocols (WS-*).

Moreover, each protocol defines its own description format, UPnP uses an XML proprietary format while IGRS and DPWS use the Web Service Description Language (WSDL). Plug-n-Play protocols also define the syntax description using standard profiles per device type. For example, a standard UPnP light [UPnP 03] profile hosts a `SwitchPower` service with a `Switch(true/false)` action to control the light while a DPWS light [SOA4D b] profile uses the equivalent action `SetTarget (On/OFF)`. The syntactic heterogeneity along with the protocols layers diversity, prevent applications to use any available equivalent device on the network to accomplish a specific task.

2.4.3 Management for Devices and Applications

The diversity of devices and applications interacting and sharing common resources in the digital home increases the need of remote management operations for the three actors and the end-user. Device manufacturers frequently perform firmware and software updates to improve a functionality or fix a security hole. Telecoms operators deploy at the end-user site customized residential gateways (RGW) and set-top-boxes (STB) to deliver a "*Triple Play*" service. For example, such service includes multimedia content, Internet and telephony along with a guaranteed QoS (quality of service).

End-users can experience a low quality service at home, for example the user receives a bad quality video when watching a movie. Such bad quality is more often due to an improper configuration of the STB or the TV. Third party application providers need also to carry out management operations to remotely install their proposed applications and to allow a successful orchestration and interaction between home devices. For example a device firmware update is needed to enable the new application. Some of these operations cannot be totally automated since they depend on the LAN configuration (DHCP, DNS-Servers, IPv4/IPv6), the device type and its execution environment (OS or virtual machines), therefore a remote diagnostic and intervention are often needed.

To support remote management operations and minimize the cost of a technician intervention on the end-user site, multiple telecoms operators, device manufacturers and service providers adopted a standard wide area network (WAN) management protocol, see Figure 2.4. The standard CPE WAN Management Protocol (CWMP) [Lupton 07] (also known as TR-069) allows using an Auto-Configuration Server (ACS) deployed on the operator platform to manage a Customer Premises Equipment (CPE). A CPE profile can be supported by a residential home gateway, a STB or a VoIP phone. A CPE offers a specific data model to represent the device status, general or specific information. The management operations include auto-configuration operations,

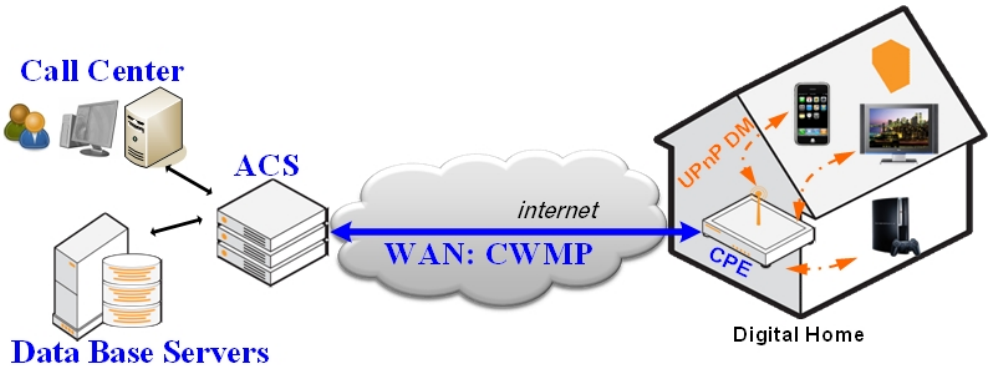


Figure 2.4: Remote Administration through CWMP and UPnP-DM

software/firmware management, such as update, download, (un)install or diagnostic actions along with status and performance monitoring to ensure a QoS fulfillment. Other standard and proprietary management protocols have emerged like OMA DM [OMA], the Simple Network Management Protocol [Harrington 02] and the web service management (WS-MAN) [DTMF] proposed by the Distributed Management Task Force (DMTF). However, it seems that the CWMP is the most adopted protocol so far [Broadband].

The CWMP scope is clearly the WAN management and targets only equipments supporting the CPE profile and capabilities, while *"Plug and Play"* protocols support the discovery, control and eventing in LANs. *Nikolaidis et al.* [Nikolaidis 07] proposed a bridge between the CWMP and the UPnP protocols to extend the operator control to non-CWMP devices in the LAN. For each UPnP device, an entry in the CPE data model is added along with its supported services, actions and state variables. The main drawback of the solution is that the operator can invoke via CWMP supported actions on the UPnP device, i.e. if the UPnP device supports only interaction commands (print, turn on light, getTime), the management operations remains limited.

To extend the management operations to the LAN, *Bottaro et al* [Bottaro 08a] proposed the UPnP Device Management (UPnP-DM) [UPnP 11] which was adopted by the UPnP Forum. The UPnP DM is a standard profile which consists of two required and one optional services. The Basic Management Service (BMS) [UPnP 10a] allows to perform basic operations like rebooting or resetting the device, performing a *"Ping"*, a *"Traceroute"* and self-tests diagnostics. It also allows to retrieve the device status and logs. The Configuration Management Service (CMS) [UPnP 10b] is designed to retrieve configuration information such as IP and MAC address, the DNS and DHCP servers IP addresses, the device CPU and memory current usage. And finally, the optional Software Management Service (SMS) [UPnP 10c] allows to manage software entities, (un)install, update, stop/start service. A proxy bridge between UPnP-DM and CWMP is proposed in [Broadband 10b] to extend the operator's management operations from the WAN to the LAN.

Management protocols adopted by device manufacturers and telecoms operators proved their efficiency to administrate and remotely configure home devices at the end-user side [UPnP 11], [Broadband 10a]. The management operations include the device and applications diagnostics, and troubleshooting. Those two operations require a historical trace of events taking place in the digital home like the appearance and disappearance of home devices, firmware and new applications deployment and management. Additionally, the network topology and its connectivity are also required to detect unplugged or malfunctioning devices, bandwidth use and other resource consumption to guarantee a QoS for the end-user experience. An historical trace of the digital home gives the call center valuable information to correlate previous events and the currently occurred failures and simplifies the diagnostic and troubleshooting operations. For example, the new installed application can be conflicted with previously installed ones or even provoke failures and crashes of other applications, or the STB is unable to connect to the TV, the trace can reveal the STB and TV device status, their link connectivity and

the last time they were connected.

The diversity of the management and the plug and play protocols forces the three actors: telecoms operators, device manufacturers and application providers to support several tools in order to provide the end-users with the required management and administration operations. Indeed, the three actors must maintain and update their multi-protocol diagnostics platforms and tools in order to provide the end-users with the necessary diagnostic operations.

Other characteristics [Kurkovsky 07], [Aipperspach 08], [Dixon 10] and challenges [Edwards 01], [Lyytinen 02], [Kindberg 02] has been pointed out in the literature, however, in this work we only focus on the dynamicity, the heterogeneity and the management aspects. We believe that an efficient middleware for the digital home must handle challenges arising from such a complex environment and provides a simple platform for the three actors and the end-users. Third party application providers will be able to offer efficient and well adapted ubiquitous applications to suit end-users needs with an easy installation. Manufacturers remotely managing devices and telecoms operators offering a guaranteed QoS along with diagnostic and management operations for devices and installed applications. And finally, for the end-user a great ubiquitous experience and a simple technology to accomplish everyday tasks.

2.4.4 The Overall Actual Device and Application Architecture

The digital home tends towards a ubiquitous environment, therefore it inherits a lot of characteristics presented in section 2.2. Consequently, a ubiquitous system handling the digital home also inherits the challenges presented in section 2.3. We provide in this section an overall layers architecture of the applications and devices of the digital home. We also rediscuss the characteristics and challenges of a ubiquitous system with regard to the actual devices and applications architecture.

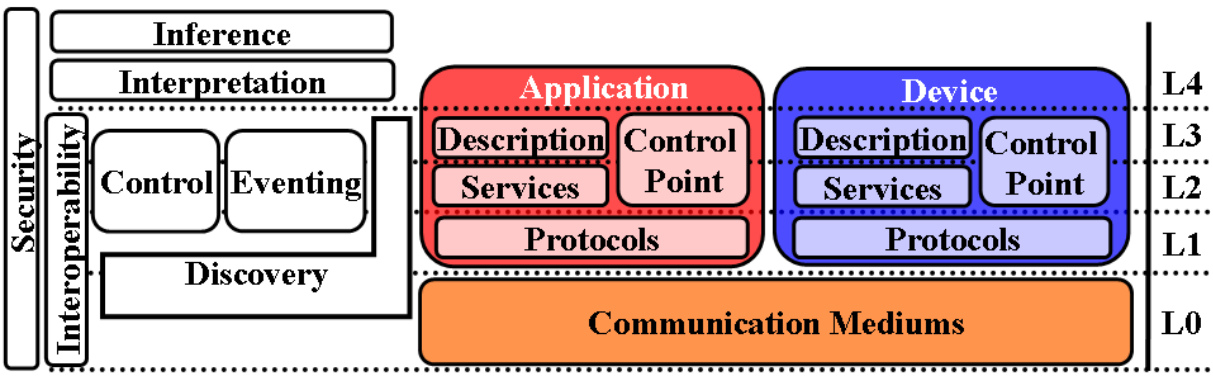


Figure 2.5: Overall Layers System Architecture

Figure 2.5 presents the scope of the ubiquitous system characteristics and challenges with regard to the current architecture of the digital home's applications and devices. It summarizes the internal architecture of a device and an application. They can be seen as entities providing a service or consuming a service. Each entity, consists of three layers. The protocol layer for exchanging information, the service or control point layer which provides or consumes a service. A ubiquitous application for example is a control point since it allows to control other devices and services. The third layer is the description layer which exposes information about the provided services and capabilities. The control point analyses the description provided by an entity to discover its capabilities and how to interact with it.

L0: Communication Mediums Each device or application interacts with other devices and/or applications. Therefore, the interaction between entities goes through a communication medium. The communication

layer (L0) is used as a medium to transport information between entities like devices and applications.

L1: Protocols Protocols at the layer (L1) are used on top of the communication medium to exchange information between two or more entities. Such protocols are embedded by applications and devices in order to be used by the provided services or the control points at the L2 layer.

L2: Interaction An entity can either be acting as a control point, as a service provider or both, situated at the L2 layer in Figure 2.5. Control points only use and coordinate capabilities provided by other applications and devices. For instance, the "Home Share Application" acts as a control point and orchestrates Bob's devices in order to allow a media sharing across the home. Devices also act as a control point, for example a TV would only control a media device like a Network Area Storage to retrieve media contents. Devices and applications can also provide one or more services. In section 2.1, Bob's home application interacts with a weather forecast application which provides the current temperature and the weather condition. Devices also offer services, for example a light device provides a `Switch` service with two different actions, the `SetTarget` to remotely control the device (turn on/off) and the `GetStatus` to retrieve its actual state.

L3: Description The description layer is where the entities providing services announce a general description about the device information and its supported capabilities. The description involves the actions invocation behavior along with the means of interaction with the provided capabilities. It also exposes the actual device state. Thus, control points interpret the description provided by the applications or devices in order to invoke their capabilities.

The service and device description are essential for discovery and interaction as mentioned before in section 2.3. It allows other devices and applications to identify the device along with its supported services and provided functionalities. Moreover, the description details the input, output parameters and formats for the action invocation and event notifications.

L4: Data Analysis The final layer is on top of all these previous layer. The data collected from the applications and devices are analyzed in the final layer to efficiently predict the user's behavior and preferences.

2.4.5 Discussion Around the Ubiquitous System Characteristics

Figure 2.5 overviews the scope of the ubiquitous system characteristics and challenges as presented in section 2.3. This section sets the scope of each characteristic with respect to actual device and application architecture.

Discovery The device discovery, as defined previously, is the ability to identify a device along with its capabilities and provided services. The discovery phase is involved in almost all the layers of the device and application architecture, L1, L2, L3. Some might argue that the device discovery should be limited to the protocol layer while others insist that the protocol layer choice is highly dependent on the communication medium L0. Indeed, the protocol is dependent on the communication medium since some of the IP based protocols are not suitable in terms of efficiency on a low range and data transmission communication mediums. Therefore, the discovery characteristic in Figure 2.5 is placed partially in the L0 communication medium layer. The device and application discovery is obviously dependent on the protocol stack used by the entity to announce its arrival or departure on the network. Moreover, the entity identification is dependent on its announced description containing the provided services and supported capabilities. The provided functionalities constitutes a major criteria whether an entity is selected to interact with or not.

Control and Eventing The control and eventing depend on the discovery. The interaction and the notification between applications and devices take place once the entity is identified along with the supported

services and functionalities. These two characteristics depend also on the service description since it details the behavior and interaction mean when an action is invoked or when a notification has been triggered. The description includes the format of input and output parameters along with the notification delay and format. The scope of these two characteristics ranges from L1 to L3. Obviously, the control and eventing also depend on the underlying protocols which allow to exchange control and eventing messages.

Interoperability The interoperability however is scattered over the first four layers, from L0 to L3. It is involved in the communication mediums. A lot of effort has been carried out in the "Internet of Things" domain in order to connect RFID communicating objects to the Internet through RFID-Internet [Jain 10] proxies. Such type of proxies bridges the communication between low power communication mediums and the wide web. There is a multitude of use cases pointed out such as monitoring of manufacturered objects or tracing a mail package on the globe through the web. The interoperability is also needed at the protocols layer in order to discover a device and interact with it. For instance an application embedding a UPnP protocol stack cannot be aware of the presence of a DPWS device. The interoperability at the service level is desired to allow applications and devices to interoperate independently from the protocol and communication mediums. The main limitation of the interoperability at the service level is the description of devices, applications and their capabilities. Many representation format and languages has been proposed to describe a service along with its supported capabilities. This multitude of representation makes it difficult for services to interoperate. Moreover, even if the description is expressed in the same description language and format, the syntax is different even if the semantics are the same. For instance a light service offers an action `SetTarget` to turn on or off the light while another similar light device represents the same action with the name `Switch`. The two actions are semantically equivalent, however they are syntactically different. An application or a device searching to interact with a device interprets the description syntactically and not semantically therefore the service identification and the interaction will not take place.

Interpretation and Inference Obviously, those two characteristics stand in layer L4 on top of the other layers. The interpretation analyzes (gesture, speech, movement) data provided by devices and applications in order to trigger actions and execute tasks. The inference, however is placed on top of all these layers including for example, the gesture or the speech interpretation, it takes as an input all sort of data, provided by devices, applications and users. The inference makes decisions based on some logic and statistical analyses from previously collected data. For instance, an application will be able to decide if Bob is sleeping based on the actual sensors detector information and an average of his sleeping hour time during the month.

Security the security is clearly transversal to all these layers starting from the communication mediums to the inference and decision making. In all these layers, devices and applications handle private and sensible data transmitted through communication mediums. Therefore the security is a must on all these layers.

In our work we will focus on the following characteristics: discovery, interoperability, control and eventing in the L1, L2 and L3 layers.

2.5 Conclusion

In this chapter, we presented the context of our work which is the ubiquitous environment and more specifically the digital home. We extracted in section 2.2 the ubiquitous environment characteristics and outlined in section 2.3 an ideal ubiquitous system characteristics. In section 2.4, we detailed the actual digital home's state and characteristics. Then we rediscussed the characteristics of a digital home system.

In our work we will focus mainly on the following characteristics and challenges already pointed out in section 2.4.

1. The device and service discovery.
2. The interoperability between Plug and Play Protocols.
3. The device and application management.

Obviously, these three challenges pointed out in the digital home environment must be tackled by a ubiquitous system architecture capable of supporting the following features: the discovery, the control, the eventing and the interoperability.

Thus, we present in chapter 3, the L2 service layer in a service oriented architecture (SOA) vision along with the its three entities: the service providers, consumers and the service registry. Then, we mainly show how the SOA characteristics meet the ubiquitous system requirements and features. And finally, we describe a Service Oriented Framework, "OSGi" with its architecture and show how it can be used to fulfill the following ubiquitous system characteristics: discovery, control and eventing. However, only the interoperability remains unsolved by the SOA characteristics.

Therefore, in order to understand the layers retaining a complete interoperability between the plug and play devices, we detail in chapter 4, the following plug and play protocols: UPnP, DPWS, IGRS, Bonjour. Then, we draw a comparison between such protocols and detail their common and divergence points. Moreover, we enumerate the three layers retaining a complete interoperability between the plug and play devices: the protocols stacks, the description format and content heterogeneity.

Thus, in chapter 5 we point out the knowledge representation in general then we go through the service and device description (L3) languages and formats. Then, we provide three knowledge representation techniques which might be useful to resolve the plug and play description format and content heterogeneity.

Chapter 3

Service Oriented Architecture

"The Service Oriented Architecture is a set of components which can be invoked, and whose interface description can be published and discovered."

– W3C

Contents

3.1 SOA Principles **47**

3.2 SOA Characteristics meeting Ubiquitous System Ones **49**

3.3 OSGi a Service Oriented Framework **50**

3.4 Conclusion **56**

In this chapter, we detail the Service Oriented Architecture along with its entities and characteristics. Then, we report how the SOA characteristics meet those of the ubiquitous system. We also provide an overview of OSGi, an SOA framework used later in our work.

3.1 SOA Principles

The Service Oriented Architecture (SOA) according to the OASIS[1] reference model [MacKenzie 06] is a way of organizing and using different capabilities. Entities like applications or devices create simple capabilities or complex ones by combining other capabilities. Such entities can also act as control points to consume capabilities. The SOA paradigm allows to match the needs with the exposed capabilities. The key concepts for describing the service oriented architecture according to [MacKenzie 06] are: visibility, interaction and effect. The visibility allows those who need to consume, and those who offer capabilities to see each others. Each entity expresses the capability or the need through a description containing different aspects, such as the input, output, technical requirements or policies. Such description is specified by domain experts or service designers and programmers for example. The interaction involves information exchange, through messages for example, and an action invocation to accomplish a need through a capability. The effect is the result of the interaction which may result a return of information or a change of state.

Three entities constitute the service oriented architecture as shown in Figure 3.1: the service provider, the service consumer and the service registry. Those entities either provide or consume services. A service can be defined as a container of one or more capabilities. For instance a light device offers a `Switch` service supporting different actions to turn on/off a light or retrieve the actual light intensity and state.

[1] `www.oasis-open.org`

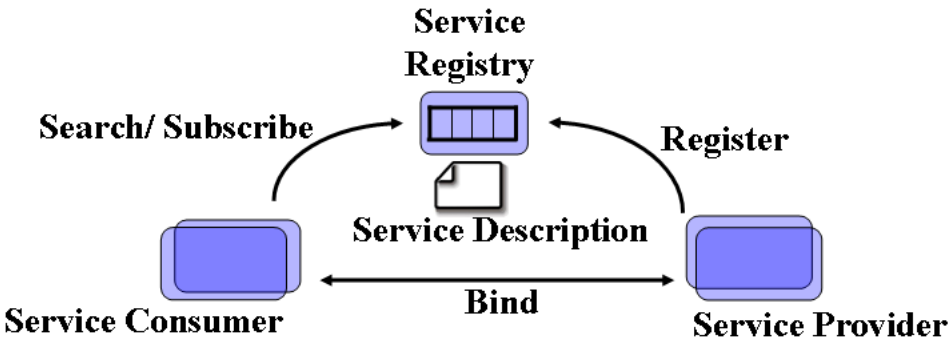


Figure 3.1: Service Oriented Architecture paradigm

Entities like applications or devices offering one or more capabilities are called Service Providers, see Figure 3.1. Ubiquitous applications or devices, acting as control points searching to consume or to interact with services are called service consumers. The service registry is an entity used by service providers to publish their service description and by the clients to look up available services.

Service providers publish their service description in the service registry. Service consumers search in the registry for a defined service to accomplish a given task. A service consumer can also subscribe to a service, it will be notified when a service provider meeting its needs appears.

Two modes are possible to discover a service, the active discovery is when the provider is available and the request is satisfied instantly. The passive discovery is when the provider is unavailable, however, the request is saved in the registry. As soon as a provider appears satisfying the request, the consumer is notified.

The service registry matches the service description with the service consumer request and returns a reference to the consumer about the found services. The service consumer receiving the service provider reference, binds to it and interacts with its provided actions. The service registry can be represented by real physical centralized or distributed entities. The OASIS proposes the UDDI [OASIS 04] protocol and registry platform for service registration and search. A service registry can also be distributed in the Local Area Network.

In a distributed registry service in the local area network scope, devices announce their services on the network while other devices and applications search or listen for the device annunciation. Once the service is found, the consumer binds with the service provider and interacts with it. Once the service provider becomes unreachable or disappears, then, the service registry removes the service description and the service provider reference. The service consumers are also notified when a service disappears.

The service description provides information about the service and its functionalities independently from the underlying implementation. It represents an abstraction of the existing invocation mechanisms supported by the service. Each service description is defined by an interface and a signature for each provided action. It details the means of interaction with the provided service along with its actions. It includes the information to be provided to the action along with the returned values and the effect of the interaction. Moreover, the interface and the description information are represented syntactically in a format that can be interpreted by the service consumer [MacKenzie 06].

As mentioned in [MacKenzie 06], there are theoretic limits on the effectiveness of the syntactic description. There will always be an ambiguity or an unstated assumption used by the service provider when defining the service interface and description. The service interface and other specific properties are used to register and request services in the service registry. The service client must know in advance the interface name of the requested service. Since the request is based on the interface name, a client requesting an interface with a semantically equivalent but syntactically different name will not receive the reference of a similar service. The

limitation of the SOA approach resides in the matching of the requested services and provided ones. Such limitation is due to the syntactic service description which can be ambiguous.

3.2 SOA Characteristics meeting Ubiquitous System Ones

The SOA characteristics has been largely discussed in the literature, like in [MacKenzie 06, Bottaro 07b, Tigli 09b]. We overview some of these characteristics and describe how they meet the ubiquitous system ones.

SOA services are loose-coupled since the dependency between services is minimized and only holds to the service interface. Indeed, the SOA services follow the separation of concerns concept between different entities, and the use of functionalities through interfaces. The loose coupling increases the services re-usability. The description provides information about the service behavior and how to invoke its actions. Such abstraction makes the interaction implementation independent since the action can be invoked according to the service interface. Discoverability is one of the main characteristics of SOA. Each service announces its description and the means of interaction through its service interface description. Such description is published in a service registry to be discovered dynamically at run-time by the service consumers. The loose coupling and the re-usability allow service composition. Service provider can offer services relying on many other services as shown in Figure 3.2 where the service C depends on the two services A and B.

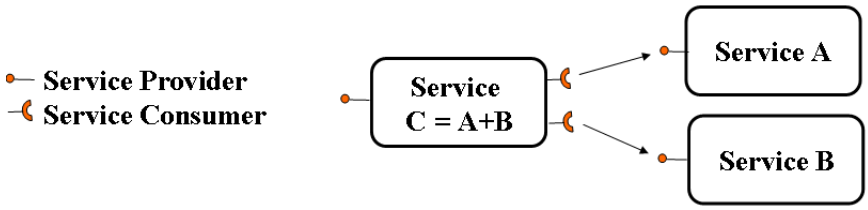


Figure 3.2: Service Composition

Many works in the literature such as [Tigli 09b] and [Bottaro 08b] acknowledge that the Service Oriented Architecture is well suited to handle dynamicity and heterogeneity of ubiquitous environments. Multiple protocols follow the SOA paradigms like UPnP [UPnP] and DPWS [OASIS 09a]. Devices are like SOA services, they announce their description on the network. Such description holds information about their supported capabilities along with the interface detailing how to invoke their provided actions. Since the devices announce their description on the network, other devices or applications scanning the network can discover and identify the device along with its capabilities in order to interact with it.

The SOA characteristics contains three of the ubiquitous environment characteristics. The discoverability feature of SOA allows to handle the device dynamicity in ubiquitous environments. SOA services announce their description and their supported capabilities to be discovered by other services. The service registry handles the service registration, it allows service providers to publish their description. The service registry also handles the service departure, it removes the service description the services disappear.

Consequently, a ubiquitous system following the SOA paradigm can actually handle the dynamicity of devices and applications if such devices also follow the SOA paradigm. In other words, if devices announce their departure/arrival on the network along with their description. A ubiquitous system can discover such devices and handle their registration to allow ubiquitous applications to interact with.

Moreover, the SOA has proved its worth to deal with applications interacting with real physical devices. Such interaction is made possible thanks to the eventing mechanism suggested by the Event-Driven Architecture (EDA) [Michelson 06] which is a part of the advanced Service Oriented Architecture. The EDA introduces a

new kind of interaction between services, real devices and software applications. Such interaction is needed in the ubiquitous environment to allow an interaction between the real physical world and the ubiquitous system and applications.

At last, the SOA provides some kind of interoperability between services. Actually, the SOA allows an interaction between services independently from the implementation. The interface represents the service facade which describes the service behavior and effect. Thus, the service capabilities are invoked based on the method signature expressed in the interface description. Therefore, the interaction is dependent on the service description.

Discovery, control and eventing are the main features of the service oriented systems that suit at best so far ubiquitous systems characteristics. However, one of the main constraints still resides in the service composition and substitution since such operations highly depend on the interpretation of the service description and interface.

We outline next, OSGi an service oriented framework.

3.3 OSGi a Service Oriented Framework

The OSGi framework is a service oriented architecture framework. It has been proposed by the *Open Service Gateway initiative* [OSGi] alliance in 1999. The OSGi alliance is an industrial consortium joining effort to specify and promote a residential gateway framework. Such framework hosts multiple services on the client residential gateways such as TV decoders and other TV services. The success of the first version attracted many firms to join the alliance. Thus, the OSGi framework enlarged its scope to become an SOA framework supporting generic service deployment and execution.

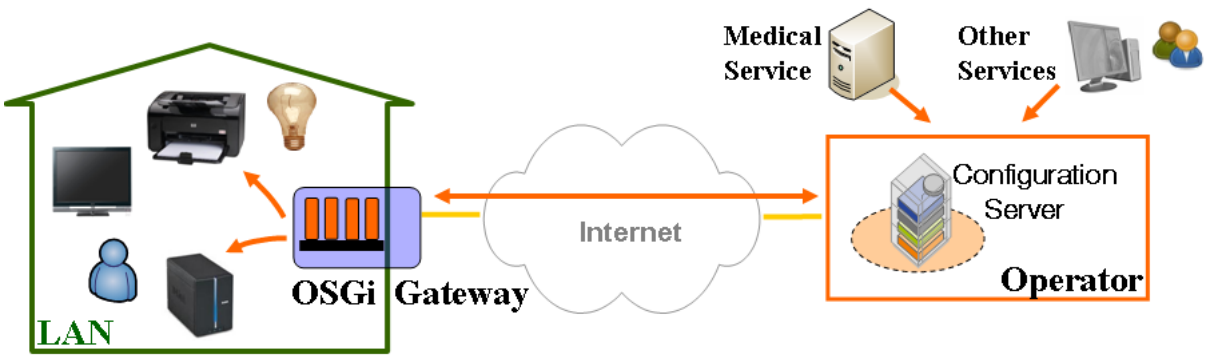


Figure 3.3: OSGi on residential gateways

Figure 3.3 shows the main motivation of the alliance to propose a dynamic modular architecture framework to be deployed on residential gateway devices. The aim of OSGi is to deploy, for example on residential gateways, a mediator between the Internet and the local area network, i.e. the digital home. Multiple devices exist in the digital home and intercommunicate to accomplish a given task for the user. However, such devices usually communicate in the local scope of a network. The residential gateway allows to connect such devices with other remote applications in order to provide an added value services to the end-user. For instance, surveillance camera devices and sensors located in the digital home can be connected to a remote monitoring system for intrusion detection and security surveillance. Additional services can also be proposed to the end-user based on to his already existing devices and applications in his home. For instance, the ubiquitous "Home Share Application"

will enable the user to share the media content across his devices. Such applications can therefore be remotely installed on the residential gateways already connected to the Internet. Telecoms operators providing such residential gateways to the end-user will manage and administrate such devices. Moreover, third parties can offer applications to the end-user such as medical assistance or security surveillance services. The third parties service providers propose such services to the end-users by deploying services on their residential gateways. Thus, the Telecoms operator share the gateway with the third parties applications to offer the end-user various services ranging from multimedia to the surveillance and health care applications.

Modularity and loose-coupling constitute the main advantages promoted by the OSGi framework. In OSGi, each service is independently packaged and executed on the framework. Moreover, the framework offers a management life cycle for each service. This feature allows a remote service management and deployment by both third parties and Telecoms operators.

3.3.1 An Architectural Overview of OSGi

OSGi is a dynamic module system based on the Service Oriented Architecture. An OSGi implementation has three main elements: the framework, the bundles and the services. As shown in Figure 3.4, the OSGi framework is an execution framework placed on top of a Java Virtual Machine (JVM). Deployment units referred to as *Bundles* are installed on the OSGi framework. Once started, a bundle implementation either offers a service, consumes a service or simply provide an API for other bundles. The services cooperate at the OSGi framework and can also interact with external services or native OS APIs.

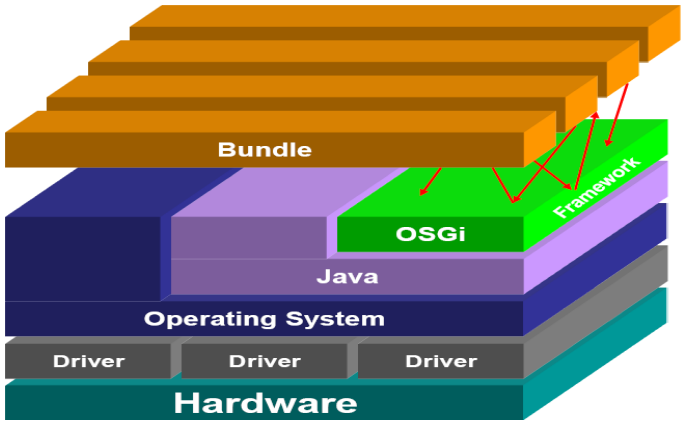


Figure 3.4: Multi Layer Architecture of an OSGi Framework [Richard S. Hall]

From the conceptual point of view, the OSGi framework has three layers [Hall 10] defined in the OSGi specification:

- The Module Layer defines the packaging and code sharing.
- The LifeCycle Layer defines the execution model management in the framework.
- The Service Layer defines the publish, search and interaction model based on the service oriented architecture.

3.3.1.1 The Module Layer

This layer defines the bundle concept which is a basic deployment unit. As shown in Figure 3.5, a bundle contains Java binary classes and other resources such as files, images or native APIs packaged in a Java ARchive (.jar)

file. Each JAR file also packages a *manifest* file containing *metadata* about the bundle. The OSGi manifest holds different information about the bundle such as its name, version, author and vendor. The manifest also expresses the bundle package dependency imported from other bundles. It also exposes the exported packages shared on the framework and can be used by other bundles. Specifying the bundle dependency allows the OSGi framework to manage the bundle resolution automatically. The manifest also contains the *Activator* class name which is equivalent to the main class in a standard JAR file. The *Activator* class allows to the OSGi framework to handle the module life cycle by invoking its specific methods start and stop as shown in section 3.3.1.2.

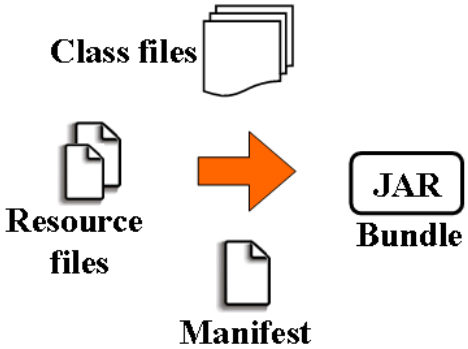


Figure 3.5: OSGi Bundle

The bundle is deployed on an OSGi framework, it can implement either a service client, a service provider or simply an API providing packages to other bundles. The bundles interact with each others at the service layer.

3.3.1.2 The LifeCycle Layer

This layer defines the bundle management in the OSGi framework. It provides a well defined life cycle for the bundles at the download, install and execution time. Such life cycle allows to dynamically manage and evolve the bundles on the fly without the need to restart the OSGi framework.

The OSGi lifecycle state diagram is shown in Figure 3.6. The mechanism offers different operations to manage dynamic installation, start, stop, update, removal and resolution of dependencies between bundles. We overview in the following those commands.

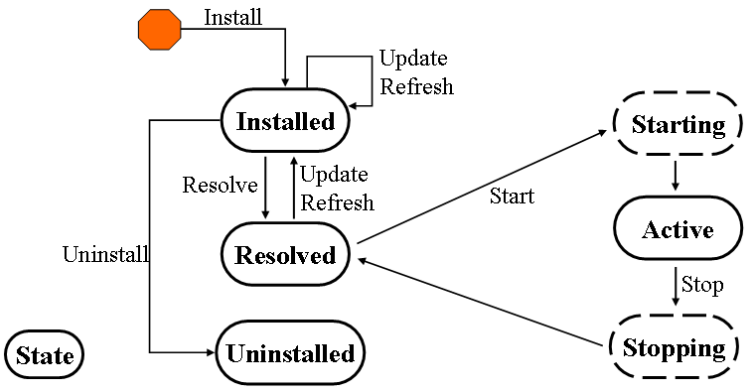


Figure 3.6: OSGi LifeCycle State Diagram

Install This operation allows to install a bundle (.jar) on the framework. The installation takes as input the JAR location and deploys the JAR file in the framework's local file system. This step places the bundle in the `INSTALLED` state. The OSGi framework provides the method `InstallBundle(String URL)` to install bundles, it takes as input the url location of the JAR file.

Dependency Resolution Once a bundle is installed, the OSGi framework automatically proceeds to resolve its dependencies. Such dependencies represent the imported packages list expressed in the bundle manifest. If the imported packages are found on the platform, then the bundles providing those dependencies are already installed and resolved. Once the dependencies are satisfied, then the framework exports the packages of the bundle and its state goes from `INSTALLED` to `RESOLVED`. The framework ensures that all the dependencies of a bundle are satisfied before it can be used. The framework will not allow a bundle to transit into a `RESOLVED` state unless all its dependencies are satisfied.

Start The Start operation allows a bundle to be started. The `STARTING` state is a transitory one, the bundle implicitly goes from the `RESOLVED` state into the `Active` state. The `Start()` method executed on a bundle object allows it to transit to the `Active` state upon successful completion of the method execution. However, if the execution throws an exception, it transitions back to the `RESOLVED` state.

Stop An `ACTIVE` bundle can be stopped suing the `Stop()` method. Its state goes through the `STOPPING` state into the `RESOLVED` state. The `STOPPING` state is a transitory one like the `STARTING` state. A stopped bundle goes back to the `RESOLVED` because its dependencies are still satisfied and there is no need to be resolved again.

Update or Refresh The refresh or the update of a bundle automatically leads to its stopping, re-installation and resolution. It is active then it will be reactivated again. If the update fails, the bundle can be either in a `RESOLVED` state if the dependencies are still satisfied or in an `INSTALLED` state if not.

Uninstall A bundle can be uninstalled, then the JAR will be removed from the framework's file system. However, some classes will still be present in the local cache of the framework if other bundles are dependent. An update or a refresh operation suppresses the remained charged classes.

The uninstall operation transits an `INSTALLED` bundle into an `UNINSTALLED` state. An active bundle can also be uninstalled. The framework automatically stops the bundle first, which transits to the `RESOLVED` state, then the framework transits the bundle state into an `INSTALLED` state before uninstalling it.

3.3.1.3 The Service Layer

The service layer details the interaction model between services following the service oriented architecture principles. Service providers publish their services into the service registry while service consumers search or subscribe to consume specific services. Once the request is satisfied, the services can bind and interact with each others.

```
//Printer Service providing two capabilities:
public interface Printing{
  public void color_Printing(file f);
  public void bw_Printing(file f); }
```

Listing 3.1: An OSGi Printer Service Example

An OSGi service provider is described by a Java interface representing an abstraction facade of its specific implementation. The service interface is used by other service consumers to interact with the service capabilities.

The service interface details the provided methods that can be invoked by a service consumer. The interaction goes through a standard method invocation provided by standard Java objects.

In the Listing 3.1, an OSGi printer service example is shown. The interface *printing* provides two capabilities through two methods. The *colorPrinting* and the black & white print.

As mentioned before, a bundle implements either a service provider, a consumer or a simply provides an API for other bundles. Once started, a service provider bundle registers its service specifying the service interface along with associated specific properties. Such properties are actually a set of a key-value objects allowing to better identify a service. The Listing 3.2 shows an example of such properties.

```
// Printer Properties
Dictionary props {
 documentType = PDF;
 printerLocation = First Floor, meeting room; }
```

Listing 3.2: Printer Service Properties Example

The bundle registers its services using the standard OSGi framework API method: `registerService("Printing", ServiceJavaObj, props)`. This method publishes the service in the register. It takes the interface service, the properties and the Java Object reference, see Listing 3.3. The `BundleContext` is a Java object inherited from the OSGi framework to allow the bundle implementation to interact with the framework and its entities such as the service registry and service listener.

```
public class Activator implements BundleActivator {
private ServiceRegistration reg = null;
private Printing printing = null;

public void start(BundleContext ctxt) throws BundleException {

printing= new PrintingImpl(); //Instantiating the service

Dictionary props=new Properties();
props.put("documentType", "PDF");
props.put("printerLocation", "First Floor, meeting room");
//Registering the service
reg = ctxt.registerService("Printing", printing, props);}

public void stop(BundleContext ctxt) throws BundleException {
if(reg != null) reg.unregister();}
```

Listing 3.3: Printer Service Implementation Example

A service consumer searching for a *Printing* service uses the method `getServiceReference("Printing")` specifying the service interface name. The search can be refined using an LDAP[2] [RFC 4517] syntax-like filter with a key-value properties. For example, `getServiceReferences("Printing","(documentType=PDF)")`. Once the service is found by the service registry, it returns the Service Java object reference to the consumer which binds to it. Then, the service consumer interacts with it according the methods signature specified in the interface. The service consumer can also subscribe to a service using a `Service Listener`. The listener will notifies the service consumer as soon as the required service provider appears on the framework.

Service Hooks The OSGi framework provides service primitives: publish, find and bind. However, such primitives do not allow services to be informed about what other services are searching for. For instance, a bundle service be aware that another service is searching for a "Printing" service or is listening to "Printer"

[2]Lightweight Directory Access Protocol

services. This information can be useful to provide an on demand service. For example, a service can proxify another service. To allow such proxification, the service must first intercept the demand then it can publish the proxy service.

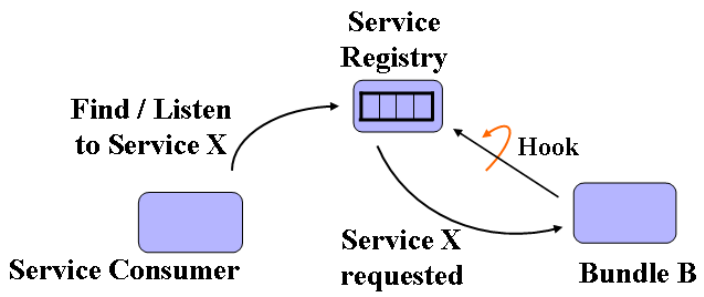


Figure 3.7: The Service Hooks

The Service Hooks specifies in [OSGi 09a], chapter 12, mechanisms allowing bundles to be informed about what other bundles are listening to or searching for. The *Find Hook* service is used by a bundle to be informed about what other bundles are searching for. While the *Listener Hook* service can be used to be informed about what other bundles are listening to. As shown in Figure 3.7, the OSGi framework will notify the bundles using the service hooks of all existing listeners. The interceptor can then use a filter to detect only specific services that are being listened for.

We applied the service hooks in our work to generate proxies on demand.

3.3.2 Base Drivers

Base Drivers are defined as a set of bundles enabling to bridge devices with specific network protocols such as UPnP, DPWS, Bonjour and IGRS. According to the OSGi Service Compendium [OSGi 09b], a UPnP base driver must provide the following functions:

- "Discover UPnP devices and map each discovered device into an OSGi registered UPnP Device service". The base driver listens to devices in the home network then creates and registers on the OSGi framework a proxy object reifying the founded device. Meaning that services offered by real devices on the home network are represented as local OSGi services with the same description. Moreover, services provided by the real device can be invoked by local service consumers on the OSGi framework. Additionally, the local invocation on the OSGi reified service is forwarded to the real device. The device reification is dynamic, it reflects the actual state of the device on the platform. Figure 3.8 shows a Printer supporting the DPWS protocol imported by the DPWS base driver and reified as an OSGi DPWS Printer. A *Print* invocation from a local service application is forwarded to the real DPWS printer by the DPWS Base Driver.

- "Present UPnP marked services that are registered with the OSGi Framework on one or more networks to be used by other computers". Service providers registered as UPnP devices on OSGi are exposed by the UPnP Base driver as real UPnP devices on the network and can be used by other services, applications or computers. Figure 3.8 shows a UPnP OSGi Light service exported as a UPnP Light on the home network.

  We used the reification in our work to expose the generated UPnP proxies as real devices on the network. Thus, the UPnP applications will interact transparently with the generated UPnP proxies as real devices.

Currently, there is a UPnP base driver implementation published by Apache [Apache b]. Multiple DPWS Base Drivers implementations has been proposed. Orange Labs, proposed through The HomeSOA [Bottaro 08b]

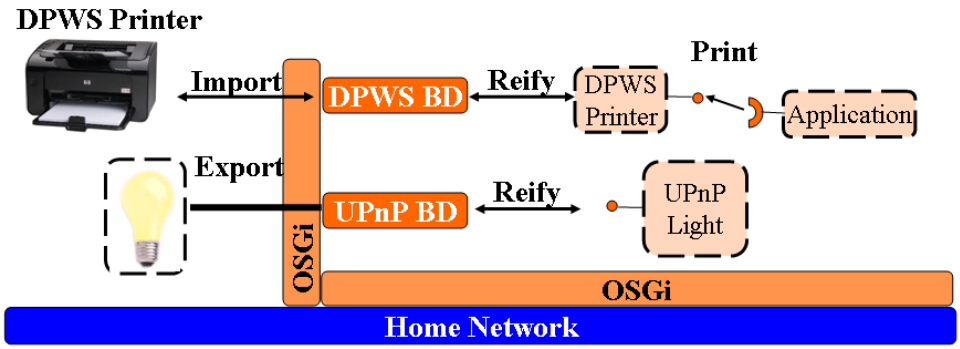


Figure 3.8: OSGi Base Drivers

project an implementation [Bottaro 07a]. Schneider electric also proposed a DPWS base driver in the SOA4D project [SOA4D a]. The WS4D [WS4D 10] project also proposes a DPWS implementation and a μDPWS for tiny sensors over 6LOWPAN [IETF 07]. A Bonjour Base Driver has also been proposed in the HomeSOA project [Bottaro 08b]. However, there is no IGRS base driver implementation to this day.

3.4 Conclusion

In this chapter we detailed the service oriented architecture along with its three entities: service providers, service clients and the service registry. Service providers publish their service information in a service registry. The published information contains description about the service such as the service provider, service behavior and capabilities. The description contains also information on how to interact with the service and invoke its functions through input and output parameters types and values. A consumer searches for services in the registry, if the requested capabilities are found, then the consumer can bind and interact with the service provider. We also showed how the SOA characteristics: the discovery, eventing and control suit at best so far ubiquitous systems characteristics.

We outlined the service oriented framework, OSGi, along with its architecture and modules. We explained how the base drivers can represent real devices on the network as local OSGi services. Thus, with the required base driver API, a reified device can be invoked locally on the OSGi framework, the base driver will handle the invocation transfer to the real device on the network. Consequently, base drivers refine the protocol stacks heterogeneity into an API and service invocations heterogeneity, since each base driver requires the use of its specific API.

Even though base drivers resolve the protocol heterogeneity, the service representation and content heterogeneity are still to be solved to allow a complete interoperability. Base drivers reify the Plug and Play devices as local OSGi services along with their supported capabilities and their description. Thus, since each protocol uses its own description format and syntax, the service description and interpretation still constitute one of the main constraints to provide a complete interoperability between services.

Chapter 4

Plug and Play Protocols

"You Plug it in, it runs, you're done"

– Combs [Combs 02]

Contents

4.1 Plug-and-Play **57**

4.2 Common features **58**

4.3 Universal Plug and Play Protocol **59**

4.4 Device Profile for Web Services **63**

4.5 Intelligent Grouping and Resource Sharing **66**

4.6 Bonjour **67**

4.7 Plug and Play Protocols Divergence **68**

4.8 Conclusion **70**

In this chapter we detail four of the plug and play protocols existing in the actual digital home. We focus only on the following IP-based protocols: UPnP, DPWS, IGRS and Bonjour. Other protocols also exist in the digital home such as the X10 [Smarthome 04] or KNX [Konnex 04] used for the home automation but are out of scope of our work.

We introduce in section 4.1 the plug and play concept, then in section 4.2, we outline the common features relation the four plug and play protocols. We detail the four protocols in sections 4.3, 4.4, 4.5 and 4.6. In section 4.7, we draw an comparison between the four protocols and show the divergence points retaining a complete interoperability between the four protocols. Finally, we conclude in section 4.8

4.1 Plug-and-Play

The *"Plug-and-Play"* (PnP) concept is defined in its early days as the automatic installation of new hardware devices by a computer system [Combs 02] such as modems, printers, network and video cards. The aim of the PnP is to simplify adding new hardware devices for novice computer users [Bigelow 99]. Therefore, the PnP allows the operating system or the driver to automatically allocate input/output addresses and memory regions for the new added device. The Plug-and-Play concept follows the three rules [Combs 02]:

- A PnP device is completely configurable by a software.
- A PnP device is capable of uniquely identifying itself to the software when required.

- A PnP device exposes to the system the required resources to operate.

The Plug and Play concept simplifying devices installation for the users encouraged device manufacturers to push this vision further. Advances in embedded systems and wireless communications allowed to establish IP based plug and play protocols following the same features as the early PnP devices. Thus, different IP based protocols has been proposed, we focus on the most wide spread:

- The Universal Plug and Play (UPnP) [UPnP] was proposed by a UPnP Forum industrial consortium in 1999 [UPnP 06b]. UPnP is the most adopted protocol so far due to the Digital Living Network Alliance (DLNA), an organism certifying that a UPnP device is conform to a standard profile and specifications.

- The Device Profile for Web Services (DPWS) [OASIS 09a] proposed by Microsoft in 2006 as an enhanced UPnP based on standard web services protocols promoted by the World Wide Web Consortium (W3C). In the summer of 2009, the DPWS became an OASIS standard. It is already deployed in Microsoft operating systems Windows Vista and Seven.

- The Intelligent Grouping and Resource Sharing (IGRS) [IGRS], a Chinese standard with a high potential of dominating Asian markets in the future.

- The Bonjour protocol proposed by Apple [Bonjour , Steinberg 05] and based on the zero-Configuration protocol. Bonjour is implemented by different Apple *i*products along with some operating systems like Windows and Unix[1].

4.2 Common features

These protocols share a lot in common, they follow the service oriented architecture (see Chapter 3) and cohabit in home networks. The term service in the plug and play domain is referred to as a container holding one or multiple capabilities represented as methods to invoke.

The multimedia domain is dominated by UPnP, IGRS and Bonjour, while the printing domain is shared between UPnP, DPWS and Bonjour. UPnP is widely adopted among routers and Wifi access points manufactures. Moreover, Plug and Play protocols support the same basic following features:

- **Addressing** is the first step when a device connects to a network, it gets an IP networking address either by searching for a Dynamic Host Configuration Protocol (DHCP) [RFC 2131] and using an allocated IP address or by assigning an Auto-IP according to the [RFC 3927].

- **Discovery and Description** occur when a PnP device joins a network, first it gets a networking address then it advertises its description on the network. The device description advertisement includes general device information such as its unique identifier, device model, type, friendly name, its hosted services and pointers to more detailed information. Ubiquitous applications acting as control points listen to the advertisement message, identify and discover the device. Then it can interact with its hosted services based on its description. A Plug and Play device also notifies its departure on the network.

- **Control** is used by control points or ubiquitous applications searching to orchestrate plug and play devices. First the control point identifies the device and browses its supported capabilities in order to interact with it. Then the service capabilities are invoked to accomplish a specific task.

[1] `http://developer.apple.com/opensource/`

- **Eventing** is the notification sent by a plug and play device to a subscribed control point. The subscription can be made on an accomplished task, a parameter update or device status change. The device notifies each subscribed control point.
- **Security** is discussed by the three following protocols: UPnP, DPWS and IGRS. However, the presence of real devices supporting security aspects remains shy. Although, the increasing demand to secure the user's data will probably promote this feature further into a real implementation and support on real devices.

4.3 Universal Plug and Play Protocol

> *"The significance of worldwide adoption of UPnP technology comes down to one simple idea: an easy to-use network becomes a ubiquitously used network."*
>
> – UPnP Forum [UPnP 06b]

The Universal Play and Play (UPnP) protocol is proposed in 1999 by the UPnP Forum which gathers different industrial companies: device manufactures, telecoms operators, companies in computing, printing, networking and home appliances. The UPnP Forum goal is to create a standard universal plug and play protocol to simplify the interaction and control between devices and applications. Currently, UPnP is the most widely adopted protocol with millions of device types and models certified by the Digital Living Network Alliance (DLNA) [DLNA 03]. **DLNA** is a **certification organism** ensuring that a device description, its hosted services and behavior are conformed to the standard specifications and profiles proposed by the UPnP Forum. A **DLNA certified** UPnP device supports the required standard behavior and description published by the UPnP Forum. The **DLNA certification** guarantees that a UPnP device description and behavior are conformed to the standard specifications. Thus, DLNA pushed the adoption of the UPnP protocol among various manufacturers and telecoms operators. Additionally, device manufacturers or vendors can also extend the device standards and propose their own services on top of the standard specifications, as shown in Table 4.1.

<table>
<tr><td colspan="4">UPnP Vendor Extensions</td></tr>
<tr><td colspan="4">UPnP Device Control Protocols</td></tr>
<tr><td colspan="4">UPnP Device Architecture</td></tr>
<tr><td rowspan="2">SSDP</td><td rowspan="2">Multicast</td><td>GENA</td><td>SOAP</td></tr>
<tr><td colspan="2">HTTP</td></tr>
<tr><td colspan="2">UDP</td><td colspan="2">TCP</td></tr>
<tr><td colspan="4">IP</td></tr>
</table>

Table 4.1: UPnP Protocol Stack [UPnP 08]

The UPnP Forum proposes the following specifications and standards to allow an efficient interoperability between UPnP devices.

- The UPnP Device Architecture (UDA) [UPnP 08] (see Table 4.1), defines the protocol stack to use for the communication between control points and devices. Table 4.1 shows the protocols over IP that a UPnP device needs to implement. The Simple Service Discovery protocol (SSDP) [Goland 99] is used during the discovery phase, the device sends its description on the network to be discovered by control points. The General Event Notification Architecture (GENA) [Cohen 98] protocol is used for the notification. The

interaction goes through the standard Simple Object Access Protocol (SOAP) [W3C 00]. The UDA also specifies a UPnP Device Template which details the description format and structure to be announced by each device on the network. The template model is shown in Figure 4.1, each device hosts one or more services, a service implements one or more actions and state variables. The actions have input and output arguments related to a state variable.

The UPnP forum defines the *StateVariable* as the argument manipulated by an action. There is an ambiguity in using the name *StateVariable* since its scope is wider than just the device or the service state. Thus, even though we will use the same terminology as the UPnP forum, however, the scope related to the *StateVariable* is wider than the just the state.

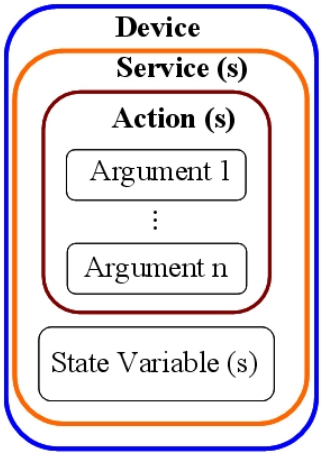


Figure 4.1: UPnP Device Description Structure

When a UPnP device joins the network, it sends an XML description file containing general information about the device manufacturer, device type and model. In the listing 4.1, the description information of a UPnP Intel Binary Light is shown in [lines (2-8)]. The UPnP Device hosts a list of UPnP Services. For each service included in the device, the description [lines (10-16)] details the service name, type and the URL location to retrieve additional information from the device.

```
1 <device>
2   <deviceType>urn:schemas-upnp-org:device:BinaryLight:1</deviceType>
3   <friendlyName>Light (G-NECVL360-10)</friendlyName>
4   <manufacturer>Intel Corporation</manufacturer>
5   <manufacturerURL>http://www.intel.com/xpc</manufacturerURL>
6   <modelDescription>Software Emulated Light Bulb</modelDescription>
7   <modelName>Intel CLR Emulated Light Bulb</modelName>
8   <uuid>5aa1392f-8408-4071-bdc3-edc513f53cf0</uuid> ...
9   <serviceList>
10   <service>
11    <serviceType>urn:schemas-upnp-org:service:SwitchPower:1</serviceType>
12    <serviceId>urn:upnp-org:serviceId:SwitchPower:1</serviceId>
13    <SCPDURL>URL to service description</SCPDURL>
14   </service>
15   </serviceList> ...
16 </device>
```

Listing 4.1: Intel Binary Light Device Description (Simplified)

A UPnP Service provides a list of UPnP StateVariables and UPnP Actions as shown in the listing 4.2.

Each action has input and/or output arguments, it allows control points to interact with or retrieve the device's state. For example, on an Intel Binary Light [UPnP 03], the UPnP Service `SwitchPower` description (see listing 4.2, lines [3-11]), includes a UPnP Action `SetTarget` with the input argument `newTargetValue`. Each action argument is related to a UPnP State Variable, for instance, on the binary light the `newTargetValue` argument is related to the state variable `Target` [line 7]. The State Variables model the state of the service and offer precise information about the UPnP Action arguments, such as the type or allowed value list. In the listing 4.2, lines [13-19] show the `Target` state variable description along with its type and default value.

A "Remote Light Control" ubiquitous application for example, listens to the UPnP devices announcements on the network, when a device description contains a `deviceType` *BinaryLight* (see listing 4.1, line [2]) is found, then the application parses the description searching for the service `SwitchPower`. If the service is found, the application can turn on/off a binary light by invoking the action `SetTarget` with a boolean value `true/false`.

The UPnP Device Architecture allows a universal interaction between UPnP devices from the networking perspective, since each device uses the underlying protocols (SSDP, GENA and SOAP) as specified. Additionally, the UPnP device behaves as expected during the addressing, discovery, control and eventing steps. As for the description, the UDA defines through the UPnP Device Template only a format and structure to represent a UPnP device, its hosted services and the supported actions. However, the device description content is not unified, i.e. any UPnP device manufacturer is totally free to choose the syntax and semantics of the service operations and behavior. For example one manufacturer could assign the type "BinaryLight" (see listing 4.1, line 2) to its device while another light manufacturer can use the type "SimpleLight". The UPnP control point or ubiquitous application scanning the local network for device announcement identifies a device through its description, therefore the difference in the device type "SimpleLight" vs "BinaryLight" syntax prevents the control points to use the required device. To allow a universal plug and play interaction, the UPnP forum proposes the Device Control Protocols (DCP) on top of the UPnP Device Architecture (see table 4.1). Thus, the DCP offers a standard device profile descriptions.

```
1  <actionList>
2   <action>
3    <name>SetTarget</name>
4     <argumentList>
5      <argument>
6       <name>newTargetValue</name>
7       <relatedStateVariable>Target</relatedStateVariable>
8       <direction>in</direction>
9      </argument>
10    </argumentList>
11  </action>  ...
12  <serviceStateTable>
13   <stateVariable sendEvents="no">
14    <name>Target</name>
15    <dataType>boolean</dataType>
16    <defaultValue>0</defaultValue>
17   </stateVariable>
18  </serviceStateTable>
```

Listing 4.2: UPnP SwitchPower Service Description (Simplified)

- The UPnP Device Control Protocol defines a standard profile description with mandatory and optional

services, actions and state variables that manufacturers need to implement. Its details for each device type the required and optional services along with the actions, their input/output arguments, data types and their range. A UPnP device conform to the UPnP Device architecture and a UPnP Device Control Protocol allows any UPnP Control Point (conform to the UDA and DCP) to interact with the device and reach a universal plug and play experience. The UPnP Forum proposed multiple standard profiles covering different domains. For example, the Home automation domain includes standard profiles for solar protection blinds, lighting controls and HVAC (Heating, Ventilating, and Air Conditioning) systems. An Internet gateway and WLAN Access point profiles are proposed in the networking field. The multimedia and printing domains also include standard profiles for a printer, camera and scanner devices. For instance, the UPnP standard printer profile promoted by the UPnP Forum under the Device Control Protocol details the printer behavior and interaction along with the supported service and actions names.

The UPnP Forum also proposes a UPnP Audio/Video Architecture [UPnP 02] to enable media content sharing on the network. The UPnP AV architecture is implemented on the same level as the DCP on top of the UDA. It defines the three following profiles: Media Server (MS), Media Renderer and an AV Control Point. The MS profile is used to describe a device capable of offering media content for example, MP3 players, DVD Players, radio. A Media Renderer profile is used to describe a device capable of playing or rendering a media, like TVs and stereos speakers. And finally, the AV Control point allows the discovery and the browsing in the media server content. The AV control point also establishes the interaction between the UPnP Media server and the Media Renderer.

Moreover, the UPnP Forum proposes other standard profiles such as the telephony, QoS, remote Access and even UPnP Low Power device to save energy devices. the UPnP Forum is currently working on the security and privacy support in the UPnP v2.

We summarize in the following how the UPnP Device Architecture allows to a UPnP device to support the Service Oriented Architecture main characteristics: discovery, control and eventing. We detail next the UPnP protocol stacks exposed in Table 4.1 which enables such features.

Discovery is the first step in the plug and play interaction. A UPnP device or a UPnP application (control point) use the Simple Service Discovery protocol (SSDP) [Goland 99] during the discovery phase. A UPnP device relies on the SSDP to announce through multicast messages its presence on the network along with its supported capabilities. A UPnP Control Point relies on the SSDP and listens to the devices' advertisements. When a device type or a capability matches, the control point requests additional information from the device. The UPnP Control Point can also uses a multicast SSDP search request. The UPnP devices listen to the search requests on the network, if a search matches the device capabilities, then the UPnP device responds with a unicast SSDP. The unicast and the advertisement messages both contains a pointer (url) to retrieve additional information on the capabilities. UPnP devices also rely on the SSDP to announce their departure messages.

Description is announced by a UPnP device on the network. It is exposed in the UPnP XML format and content detailed previously. A UPnP Control Point interprets the advertised description and if a match is found then the interaction and control can take place.

Control is supported by the Simple Object Access Protocol (SOAP) [W3C 00]. The action invocations are exchanged through SOAP messages which contain the action to invoke and the parameter(s) value(s).

Eventing is supported by the Generic Event Notification Architecture (GENA) [Cohen 98]. It allows for a control point to subscribe in order to receive notifications, for example, on a parameter change or upon

an ending task. The control points interested in receiving notifications, subscribe to an evented state variable, and provide the destination location where to send the event. The control point also specifies a subscription time which can be renewed or canceled.

The DPWS protocol is detailed in the next section.

4.4 Device Profile for Web Services

The Device profile for Web Services also known as Web Services on Devices (WSD) was initially proposed by Microsoft in 2004 as an enhanced version of the UPnP protocol. Actually, the UPnP protocol is based on non standard protocols like GENA, SSDP and uses non standard description language to announce its capabilities. Therefore, the DPWS protocol insisted on the fact that it is built on common Internet standards promoted by the *W3C*. The DPWS targets similar devices and objectives as the UPnP protocol. However, it is fully dependent on standard Web Services technology and protocols. The DPWS is seen as an extended Web Services provided by devices. Therefore, the interoperability with the existing Web Services is supposed to take place with a minimum integration effort.

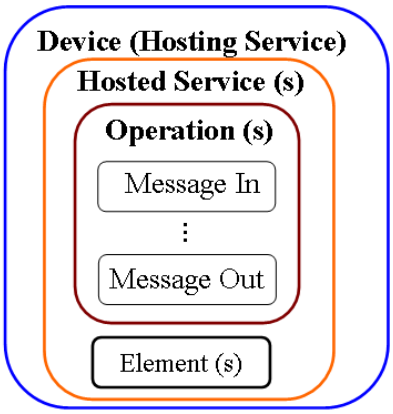


Figure 4.2: DPWS Device Description Structure

The DPWS was approved as an OASIS standard in 2009. It is already deployed in Windows Vista and Seven. Other companies are pushing this standard further, for instance, the EU Research Project SOCRADES[2] composed mainly by SAP, Schneider Electric, and Siemens, are focusing on implementing DPWS applications and devices in industrial automation systems.

The DPWS protocol provides so far only three standard profiles a printer [Microsoft 07], a scanner and video projector. Industrials like *Lifeware*[3] and *Microsoft* collaborate with manufacturers to propose DPWS-enabled devices. Furthermore, many works in the literature conducted research using the DPWS protocol. In [Quan 08], DPWS is used in a massive manufacturing system. *Parra et al* in [Parra 09] proposes a Peer-to-Peer architecture in a smart home environment where devices interoperate without a central coordination node to accomplish specific daily tasks.

The internal device description architecture is described is Figure 4.2. The device or the hosting service hosts one or multiple services (referred to as hosted services). Each hosted service (represented as "portType" in the WSDL) offers one or more capabilities which can be accessed through operations. The operation is invoked

[2] www.socrades.eu/Home/default.html
[3] www.exceptionalinnovation.com

through input messages and can return output messages. The message content contains an element which could be simple or complex and contains itself several simple or complex elements.

```
1 <in SOAP:ENV WS-Discovery>
2 <wsd:Types>wsdp:Device wprt:SimpleLight</wsd:Types>
3
4 <wsdl:definitions name="Lighting" ...
5  <xsd:simpleType name="PowerState">
6  <xsd:restriction base="xsd:token">
7   <xsd:enumeration value="ON" />
8   <xsd:enumeration value="OFF" />
9  </xsd:restriction> </xsd:simpleType>
10 <xsd:element name="Power" type="tns:PowerState" />
11
12 <wsdl:message name="SwitchMsg">
13     <wsdl:part name="PowerOut" element="tns:Power"/>
14 </wsdl:message>
15
16 <wsdl:portType name="SwitchPower" wse:EventSource="true">
17   <wsdl:operation name="Switch">
18    <wsdl:input message="tns:SwitchMsg"/>
19   </wsdl:operation> ...
20 </wsdl:definitions>
```

Listing 4.3: DPWS SwitchPower Service Description (Simplified)

The service description file is expressed using the Web Service Description Language [Christensen 01] and XML Schema Definition Language [W3C 01] to represent the content and element data type. The Listing 4.3 shows a part of a simple light [SOA4D a] DPWS device description. In the Listing 4.3, lines [16-19] defines the operation `Switch` and its input message provided by the service `SwitchPower`. Lines [12-14] defines the message content and structure by referring to the element `Power`. The element structure and content is described in lines[4-10], the type is a token having values `ON` or `OFF`. In the simple light example, the element is flat, meaning there is no other elements embedded in the `Power` element. The WSDL specification allows a hierarchical structure of elements, an element can contain more than one element. Figure 4.3 shows a part of WSDL Standard DPWS Printer description [Microsoft 07] which is shown in Figure B.1 in the appendix B.2.

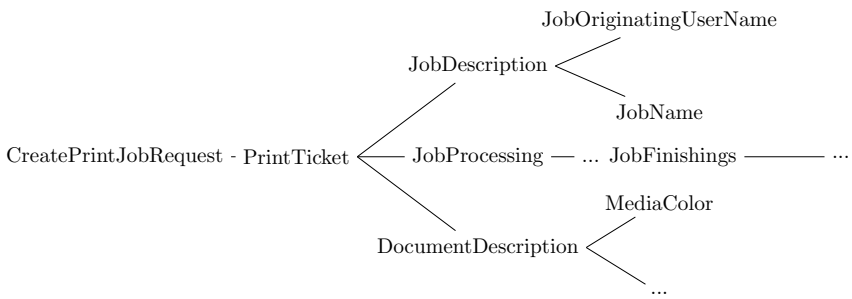


Figure 4.3: Hierarchical Parameters in the WSDL Printer description

The DPWS protocol also follows the service oriented architecture which allows discovery, control and eventing. We detail in the following paragraph, the DPWS protocol stacks exposed in Table 4.2 which enables those features. We also overview their scope in an exchange example between a client and a device as exposed in Figure 4.4.

Discovery is the first step where consumers discover devices providing services. The WS-Discovery [OASIS 09b]

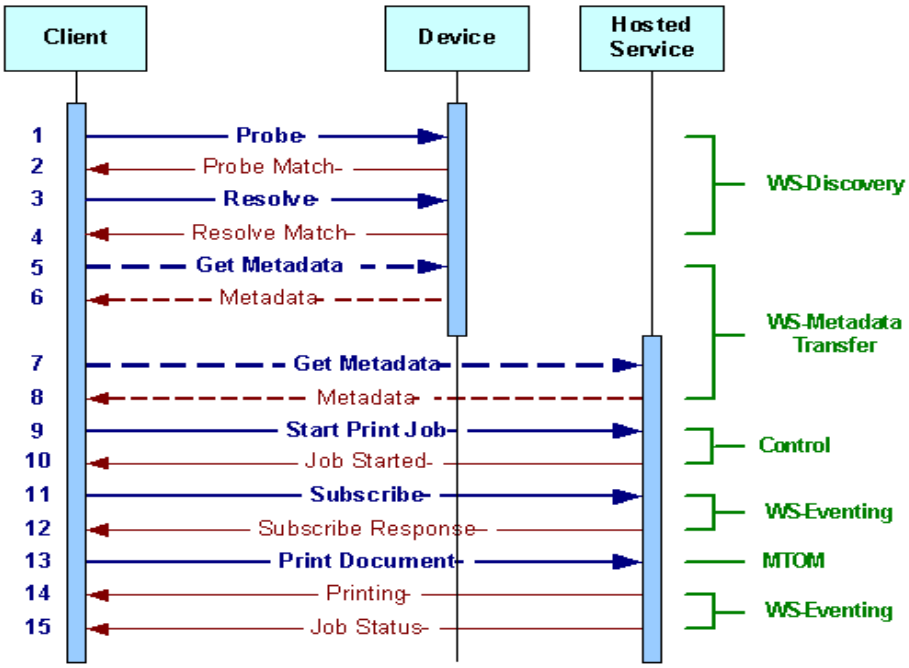


Figure 4.4: DPWS Exchange Example [Microsoft 06]

is used to discover devices. Each DPWS device announces its presence by sending a multicast message *Hello* on the network. A device also announces its departure through a *Bye* message. For the device discovery, the multicast mode is supported on the network. Additionally, DPWS specifies a discovery proxy acting as an SOA like service registry. Applications and devices know about the discovery proxy through a DHCP record [Microsoft 06]. Devices announce their arrival and departure directly to the discovery proxy. Clients search for devices in the discovery proxy to avoid generating multicast traffic on the network.

<table>
<tr><td colspan="3">Applications</td></tr>
<tr><td>WS-Eventing</td><td>WS-Discovery</td><td>WS-MetaDataExchange,
WS-Transfer</td></tr>
<tr><td colspan="3">WS-Addressing,
WS-Policy,
WS-Security</td></tr>
<tr><td colspan="3">SOAP,
WSDL, XML Schema</td></tr>
<tr><td colspan="2" rowspan="2">UDP</td><td>HTTP</td></tr>
<tr><td>TCP</td></tr>
<tr><td colspan="3">IPv4/IPv6</td></tr>
</table>

Table 4.2: DPWS Protocol Stack

Figure 4.4 shows an example of interaction. The client first sends a *Probe* message searching for a printer device. The printer device listening to *Probe* messages replies with a *probe match* message containing a WS-Addressing reference, a UUID (Universal Unique IDentifier) along with the device capabilities. The client can use the WS-Addressing [W3C 06a] reference, shown in messages [3-4] of Figure 4.4, and ask for a resolve match to retrieve more information to identify the device such as the IP address.

Description Once the device discovery takes place, the client can retrieve additional information such as the supported services and capabilities. The WS-Transfer [W3C 06c] and the WS-MetaDataExchange [W3C 11] protocols, see Table 4.2, are used to retrieve specific meta data along with the service description WSDL. The WS-Policy can also be used to express the capabilities and constraints of the Web Service. In Figure 4.4, messages [5-8] are initiated by the client to retrieve additional information about the device and its supported services and capabilities. The device responds with the meta data and the WSDL. Those messages are optional if the client knows that the device is conformed to a standard profile.

Control The interaction between devices and consumers is message based over the SOAP protocol. DPWS defines also how the message and its content are serialized during the exchange. The message format, structure and content are defined in the WSDL file, see Listing 4.3. The WS-Addressing [W3C 06a] is used on top of SOAP and involved in different features. It is used to exchange addressing information to identify a device and a service through a SOAP level address. In Figure 4.4, messages [9-10], the client starts a printing job on the device which responds that the job is started. The interaction content follows the description exposed in the WSDL file.

Eventing the notification is supported by WS-Eventing [W3C 06b]. It allows applications to subscribe to a specified event, such as when the printing operation is finished. The WSDL also contains information about the subscription and the information delivery structure and content data type. In Figure 4.4, messages [11-12], the client subscribes to the printer status during the printing job.

Messaging The DPWS uses the Message Transmission Optimization Mechanism [W3C 05] with the SOAP [W3C 00] protocol in order to exchange large amount of data such as the audio/video streams and large documents. In Figure 4.4, message 13, the client sends the document to print using MTOM over SOAP. The printer informs the client about the printer and the job status, messages [14-15].

Security The DPWS protocol offers through WS-Security a solution to exchange secured data between a device and a control point.

4.5 Intelligent Grouping and Resource Sharing

The IGRS [IGRS] alliance was established in 2003 to provide interoperability between three worlds called the "3C": Computing (laptops, PC), Consumer Electronics (TV, Set-Top-Box) and Communication (mobiles, PDAs). In 2005, the IGRS standard was formally approved by the Ministry of Information Industry of China which became the governing body of the IGRS working group. International corporations like Philips, LG, Cisco and STMicroelectronics joined the alliance transforming IGRS into a high potential standard protocol at least in the Asian markets.

The IGRS protocol stack as described in [IGRS 06], adopted the same protocol stacks used by UPnP. The IGRS standard uses GENA for eventing, SSDP for discovery and SOAP for control. However, IGRS changed the header syntax of protocol messages on the eventing and control levels. Additionally, IGRS supports a peer-to-peer discovery mode and a centralized mode (master-slave) similar to the proxy discovery mode in the DPWS protocol.

Another difference between UPnP and IGRS resides mainly in the device and service description level. UPnP uses an XML template description format, while IGRS uses the Web Service Description Language to express the device and service description. Interoperability guidelines between UPnP and IGRS are provided in chapter 11 of [IGRS 06]. UPnP devices can be discovered by IGRS control points and IGRS devices can be discovered by UPnP control points. *Xie et al* proposed in [Xie 09] a simple technical solution for syntax header

conversion to allow UPnP and IGRS device discovery and interaction. In [Bottaro 06], coexistence concerns has been pointed between UPnP and IGRS since both protocols use the same multicast address. The coexistence has been made possible, since IGRS accepted a multicast modification for the IGRS device discovery.

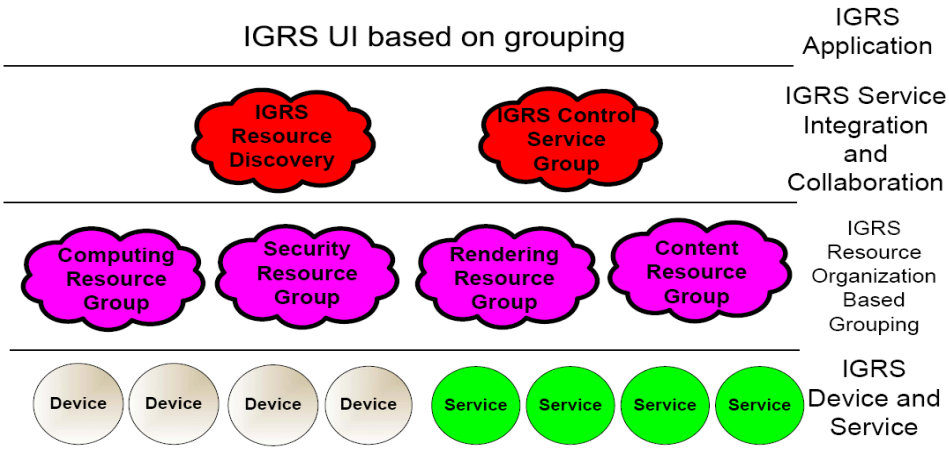


Figure 4.5: IGRS System Architecture

The IGRS proposes a "Device Grouping" concept showed in Figure 4.5 which handles the negotiation among different devices and organizes groups around specific resources such as computing, security or content. Thus, an IGRS control point can search by device category instead of searching only according to a device type. Interaction between the groups is enabled and goes through the security resource group which is present at the control and transport levels.

The IGRS labs publish standard IGRS profile devices highly inspired from the UPnP standard specifications. For instance, there is an IGRS standard profile for the set-top-box, mobile phone and even an IGRS AV [IGRS 07] (Audio, Video) specification similar to the UPnP AV. IGRS devices seem to be commercialized in Asia but not in Europe.

IGRS relies on the same UPnP protocol stacks, it also supports the control, eventing and control.

4.6 Bonjour

The Bonjour protocol (formerly known as Rendezvous [RFC 3927]) is the Apple implementation of the *Zero Configuration Protocol* [Steinberg 05] proposed by the IETF. Bonjour is already supported in the Apple products such as *i*Tunes and *i*P* devices. Some printer and network storage devices also support Bonjour.

Bonjour is based on the already widely deployed DNS protocol and server. Its is involved in the addressing, naming and service discovery. The multicast-DNS [Apple 11b] handles the addressing and the naming while the DNS Service Discovery [Apple 11a] is used to advertise and discover services.

Devices hosting Bonjour announce their hosted services in conformance to the following format *Device-Name._protocol._transportProtocol.Domain port number*.

For instance, an Apple Bonjour printer would announce the following service *Apple LaserWriter 8500._ipp._tcp.local. Port 631*. The DeviceName is a unique name on the local network. Bonjour specifies how the device should proceed to resolve conflicting names on the same local network. A Bonjour device must also announce its supported protocol and type. Table 4.3 shows an example of a printer service announcement capabilities. As Bonjour is only used for addressing, discovery and announcement, other protocols are used for the interaction and must be specified in the service announcement in the *protocol* field.

Protocols	Service Type
LPR	_printer._tcp
IPP	_ipp._tcp
AppSocket PhaserPort JetDirect Port 9100	_pdl-datastream._tcp

Table 4.3: An Apple Printer Supported Services Example [Apple 05]

Additionally, the DNS-SD allows to provide additional information during the annunciation. Such information are stored in a DNS TXT records in a key-value pairs. However, the information provided by the TXT records is considered as inferior and less reliable than those advertised initially. For each device profile, there is a standard description key record. For example, the Bonjour printer service [Apple 05] defines a set a standard keys such as product, name, pdl. The Listing 4.4 shows an example of an LPR TXT record of PostScript printer.

```
1 .txtvers=1.rp=auto.qtotal=1.priority=25.ty=Apple LaserWriter 8500
2 .note=:adminurl=http://LaserWriter8500.local./rendezvouspage.html
3 .product=(LaserWriter 8500).pdl=application/postscript
4 .Punch=3.PaperMax=legal-A4
```

Listing 4.4: LPR TXT record for a PostScript printer [Apple 05]

Bonjour only supports the service discovery and annunciation. Thus, other protocols are used on top of Bonjour for interaction and control. There is no unified control protocol, the interaction protocol used is specific to the device domain and type. For instance, the Media Server *iTunes*(v4.2) uses the Digital Audio Access Protocol (DAAP[DAA 05]) and the Digital Media Access Protocol (DMAP), while the printers use multiple interaction protocols: IPP (Internet Protocol Printing) [RFC 2567] or/and LPR (Line Printer Daemon Protocol) [RFC 1179]). Each interaction protocol has its own description format, in LPR for example, the description is ASCII-based, set in TXT LPR records[Apple 05]: *product=(LaserWriter 12/640 PS)*, *adminurl=http://printer.local./path/configpage.html*. DAAP uses another description to exchange information between iTunes (media server) and a client requesting a list songs, for example, *daap.songformat*, *daap.songartist*.

Thus, Bonjour only supports addressing, announcement and discovery and relies on other protocol layers such as the LPR or DAAP for control and eventing.

The next section draws a comparison between the four plug and play protocols.

4.7 Plug and Play Protocols Divergence

> *"Increased homogeneity in the domestic environment plainly offers attractions such as convenience. ... However, this is a double edged sword, resonating with concerns of McDonaldization, the process by which modern society takes on the characteristics of a fast-food restaurant."*
>
> – Aipperspach et al. [Aipperspach 08]

UPnP [UPnP], IGRS [IGRS], Apple Bonjour [Bonjour] and DPWS [OASIS 09a], cohabit in home networks and share a lot of common points. They are all service-oriented with the same generic IP based layers:

addressing, discovery, description, control and eventing. They also target similar device types: multimedia is shared between UPnP, IGRS and Bonjour while the printing domain is dominated by UPnP, Bonjour and DPWS.

Plug and Play Protocols				
	UPnP	DPWS	IGRS	Bonjour
Discovery	SSDP	WS-Discovery	SSDP	mDNS, DNS-SD
Service Repository	multicast	Proxy Discovery, multicast	Master-Slave, multicast	DNS Server, multicast
Control	SOAP+MTOM	SOAP	SOAP	–
Eventing	GENA	WS-Eventing	GENA	–
Security	UPnP Security	WS-Security	IGRS Security	–
QoS	UPnP QoS	WS-Policy	–	–
Description	XML-UPnP	WSDL	WSDL	Apple-Format

Table 4.4: Plug And Play Protocols Stack Comparison

Even though those protocols have a lot in common, devices can not cooperate due to following three main differences:

Protocol Stack : As shown in Table 4.4, plug-n-play protocols define their own underlying protocols each adapted to its own environment. UPnP and IGRS use the SSDP protocol for discovery. While DPWS uses the WS-Discovery along with WS-Transfer, WS-Addressing and WS-MetaDataExchange to discover devices and retrieve additional information. Bonjour employs the multicast DNS and the DNS-SD for the service discovery and annunciation.

The four protocols support the peer-to-peer device discovery mode through multicast where devices and applications listen to the arrival and departure of devices on the network. Additionally, DPWS, IGRS and Bonjour support a centralized mode for the device discovery. In such mode, devices arriving on the network register their description in the central registry service, applications searching for devices directly query the central registry. Such mechanism allows to reduce the communications on the local network. In DPWS, the service registry is supported by the Proxy Discovery, in IGRS it is the master-slave mechanism, while in Bonjour it is handled by the DNS server.

For the control, UPnP, IGRS and DPWS use the SOAP protocol. IGRS changed the header syntax of the messages when it handles the control and the eventing. Additionally, DPWS uses the MTOM mechanism to optimize the transfer of large data between the devices and the consumers. Bonjour is used only for the device and service annunciation and discovery, thus for the other features, specific protocols depending on the device domain (printing, multimedia) are employed.

GENA handles the eventing in UPnP and IGRS, while WS-Eventing is used in DPWS. In Bonjour devices, the eventing depends on the protocol layer employed on top of Bonjour.

The security in DPWS is handled by WS-Security and WS-Policy to provide secure channels during sessions. IGRS uses IGRS-Security present in the control and eventing protocols. UPnP launched initiatives to support security, however, it was not adopted by the industry because it was too complex to implement. Another simplified security layer is expected to appear in the next version of the standard profiles. In UPnP and DPWS, the security is supported separately from the specifications while in IGRS it is directly involved in the control and eventing layers.

Device and Service Description : During the annunciation each device exposes general device information (id, manufacturer, model etc), supported service interfaces along with associated action signatures and parameters. Each of these protocols use a format description to expose its information and describe its supported services. DPWS and IGRS use the Web Service Description Language to expose the description while UPnP uses the UPnP XML template format. Bonjour uses also a limited proprietary format and dependent on the device type and domain.

Table 4.5 resumes the description concepts used between the UPnP, DPWS and IGRS protocols. UPnP and IGRS uses the concepts devices and services. While DPWS has more a web service oriented vision and therefore uses hosting and hosted services concepts. UPnP refers to the capabilities supported by a service as an action while IGRS and DPWS employs the concept of operation from the Web Service Description Language. The UPnP state variables are the parameters handled by the UPnP actions in general are flat without a hierarchical structure, except for the UPnP DM [UPnP 11] parameters. While in the WSDL, a message content can be a simple element or a complex one. A complex element is composed of simple elements and complex ones, as shown in Figure 4.3.

Thus, the structural difference in the representation format adds another layer of difficulty when its comes to provide interoperability between description devices and services.

Concepts	Device	Service	Action	$Variable_{In/Out}$
$UPnP_{xml}$	UPnPDevice	UPnPService	UPnPAction	UPnP$State\ Variable_{In/Out}$
$DPWS_{wsdl}$	Hosting Service	Hosted Service	Operation	$Message_{In/Out}$ Element
$IGRS_{wsdl}$	IGRSDevice	IGRSService	Operation	$Message_{In/Out}$ Element

Table 4.5: Device Description Concepts Comparison

Description Content : In addition to the protocol stacks and description format, the description content constitutes another layer of heterogeneity. As mentioned before, those Plug-and-Play protocols target multiple similar domains. Thus, two devices having the same type support different protocols. For instance, an HP printer device supports the UPnP protocol while another Xerox printer device supports the DPWS protocol. And even more, device manufacturers propose the same device type with different plug and play protocols.

Even though, two equivalent device types support the same basic functions, the content is expressed syntactically differently. Thus, the functions are semantically equivalent but expressed in different syntax. For instance, on a standard UPnP printer, the action `CreateURIJob` is equivalent to the association of two actions `CreatePrintJob` and `AddDocument`. Another example can be found between the UPnP and DPWS lights. On a DPWS light [SOA4D b], `Switch(Token ON/OFF)` is semantically equivalent to the `SetTarget(Boolean)` on a UPnP light [UPnP].

This heterogeneity prevents smart applications to use any available device, regardless of their protocol, to accomplish a certain task such as printing or dimming a light.

4.8 Conclusion

In this chapter, we detailed the following Plug and Play protocols: UPnP, DPWS, IGRS and Bonjour.

The UPnP protocol is the most widely spread so far. The DLNA organism which certifies UPnP devices and ensures their conformity to the standard profiles allowed the wide adoption of UPnP by a large number

of manufacturers and Telecoms operators. Furthermore, to this date, UPnP is the protocol that offers the highest standard profiles among other protocols. It proposes various standard device profiles ranging from simple devices like binary lights to more complex devices like printers and scanners.

The DPWS protocol proposed by Microsoft and standardized in 2009 is already deployed in Windows Vista and Seven. Moreover, printer manufacturers adopted the printer standard profile and already propose various DPWS-capable printers. A DPWS device uses a set of (WS-*) standardized protocols and expresses its information and services with the Web Service Description Language. Such conformance with the web services protocols and description allows DPWS devices to communicate with Web services clients. To this date, DPWS published only a printer and a scanner standard profiles.

The Bonjour protocol proposed by Apple, is already deployed in *i*Tunes and the *i*P* products. It uses the multicast-DNS and DNS-SD protocols, therefore, it benefits from the large deployment of the DNS protocol and servers. However, Bonjour is only used for device addressing and discovery. Another set of protocols are used on top of Bonjour, depending on the device domain to allow eventing and control.

The IGRS protocol has a lot of similarities with UPnP and it is already emerging in Asian markets.

We also presented in this chapter, the common features and divergence points between the plug and play protocols. We pointed out in detail the main challenges restricting the interoperability between those protocols: the protocol stacks, the device and service description, and finally the content description.

Those plug and play protocols are either proposed or carried out by giant corporations. Thus, devices supporting UPnP, DPWS and Bonjour are already spread at the globe scale. Moreover, IGRS is emerging in Asian markets. Therefore, the interoperability between the four protocols is essential and constitutes a major challenge for different actors.

Applications Designers specify and implement applications to interact with various devices present in the digital home to accomplish specific tasks. Supporting multiple protocols is time consuming from the developers perspective since they must implement the interaction with each device.

Telecoms Operators deploy Internet gateways for the end-user and share the platform with third parties applications providers. They also provide tools to troubleshoot and diagnostic devices and applications to avoid an expensive human intervention at the end users side. Thus, such protocol diversity makes the diagnostic and the troubleshooting a more complex operation and adds a burden costs on their infrastructure and tools to target additional protocols.

End Users can be frustrated since they will be restrained to a certain protocol and technology if they want a complete interoperation in their home network.

Indeed, the OSGi framework along with the base drivers allow to resolve the protocol stacks heterogeneity by representing each device and its services as an OSGi service on the network. However, device and service descriptions and content remain heterogeneous.

Chapter 5

Knowledge Representation

"Unlike the human mind, computers do not have such a transparent mechanism for acquiring and representing knowledge internally, just by themselves. They rely on humans to put the knowledge into their memories."

– Gašević *et al* [Gašević 06]

Contents

5.1 Ontologies **75**

5.2 Rules **79**

5.3 Models **81**

5.4 Conclusion **82**

As detailed in chapters 3 and 4, plug and play devices and services announce their description in a service registry or on the network. Such description contains information on their supported capabilities and how to interact with their provided actions. The information provided by such services is a knowledge shared between service providers and service clients searching for specific services to use. Service consumers request and interact with the service providers according to their capabilities expressed in their description. Therefore, the description representation by the service providers and the correct interpretation by the service clients are essential for the interaction between services.

In the comparison of chapter 4, we detailed the main elements preventing plug and play device interoperability. It is due to the heterogeneity of the three following elements: the protocol stacks, the format description and the description content. Chapter 3 presents a technical solution to wipe off the protocols heterogeneity by combining the OSGi framework along with specific network base drivers. The presented solution reifies plug and play devices on the network as services on the *OSGi* framework. Thus, it allows applications and other services to interact with the reified *OSGi* services independently from the networking protocol. The solution resolves the protocol stacks heterogeneity, however, the service description and content heterogeneity remain unsolved.

In order to resolve such heterogeneity between representations, we must first have a look on the knowledge representation techniques. Therefore, in this chapter, we first define the knowledge representation. Then, we provide an insight of some representation techniques such as the ontologies, the rules and models.

"What Is a Knowledge Representation?" Davis *el al* [Davis 93] answer this question by defining five different roles of a knowledge representation. The first role is a **substitute** of a thing from the real world in an intelligent entity. The representation captures some approximation of the reality, thus it is inevitably incomplete. The second role of a knowledge representation is a **Set of Ontological Commitments** about the reality. Since, a representation is imperfect, then it is about making choices to determine what an intelligent entity "captures" from the real world and what other aspects are ignored. The third role is a **Fragmentary Theory of Intelligent Reasoning**, in other terms, what information can the system extracts from what it already knows. The reasoning can be based on rules and logics, and it can also benefits from the user's lead or recommendations. **A medium for efficient computation** is the fourth role, information should be organized in a way for the computation and the reasoning to be the most efficient. And finally, a knowledge representation is also a **medium for human expression**, humans gather and store knowledge in intelligent entities. Thus, humans have to create medium of communications, to communicate representations to machines.

Humans define the devices' descriptions which represent a knowledge to be shared with other applications and devices. The description expresses such information in various formats and languages. For instance, UPnP uses an XML template format, while IGRS and DPWS use the Web Service Description Language. As pointed out by Davis *el al*, the representation language is the fifth role which arises questions mainly about expressivity. The languages used in the plug and play protocols, allow to mainly capture the description syntactically. In the description structure, some semantics can be retrieved such as the nature of the used entities, such as device or service. However, there is no direct semantic properties pointing out the relation between the entities. Thus, the expressivity of the languages used in the plug and play protocols lacks of expressivity since it relies only on the syntactic information representation.

Moreover, each protocol defines specific profiles such as light, printer, clock with different representation but similar semantics. For instance, on two light devices, the `Switch` and the `SetTarget` actions are syntactically different but semantically similar since they accomplish the same basic task which is turning on/off the light. Thus, as pointed out in the first two roles of *Davis et al* [Davis 93] definition, the description content captures the reality which is clearly perceived almost equivalently from the semantic point of view. This lack of semantic description adds another layer of non expressivity to the existing plug and play description languages.

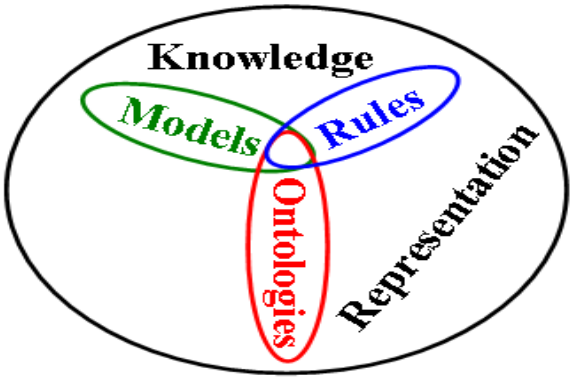


Figure 5.1: Knowledge Representation Techniques

In order, to provide interoperability between service descriptions, we outline in the following, the knowledge representation techniques. Figure 5.1 shows that the knowledge representation can be captured by three techniques:

- **Ontologies** express the knowledge semantically by defining concepts and expressing the relations among

them.

- **Rules** are usually used to reason about data. In other words, the rules are used to extract and manage data from what is already existing. However, in the knowledge representation definition proposed by Davis et *al* [Davis 93], the reasoning in one of the five roles used to represent the knowledge. Thus, the rules are involved in the knowledge representation since they allow to extract and reason from what is already represented. Moreover, the rules go beyond the knowledge representation by additionally allowing **knowledge management**. Such management is realized by extracting information to trigger actions or make conclusions on the already existing data [Gašević 06].

- **Models** enable capturing the knowledge on different abstraction layers form the real world.

These three techniques are described in the following sections.

5.1 Ontologies

The *Ontology*" comes from the Greek, *"Ontos"* for "being" and *"Logos"* for "study". It refers in philosophy to the nature of existence, the categories of being and their relations. More precisely, according to [Kalfoglou 01], *"An ontology is an explicit representation of a shared understanding of the important concepts in some domain of interest"*.

Gašević *et al* [Gašević 06] define the ontology as a language and a sort of catalog used to manipulate entities of a domain, *"To someone who wants to discuss topics in a domain **D** using a language **L**, an ontology provides a catalog of the types of things assumed to exist in **D**; the types in the ontology are represented in terms of the concepts, relations, and predicates of **L**."*

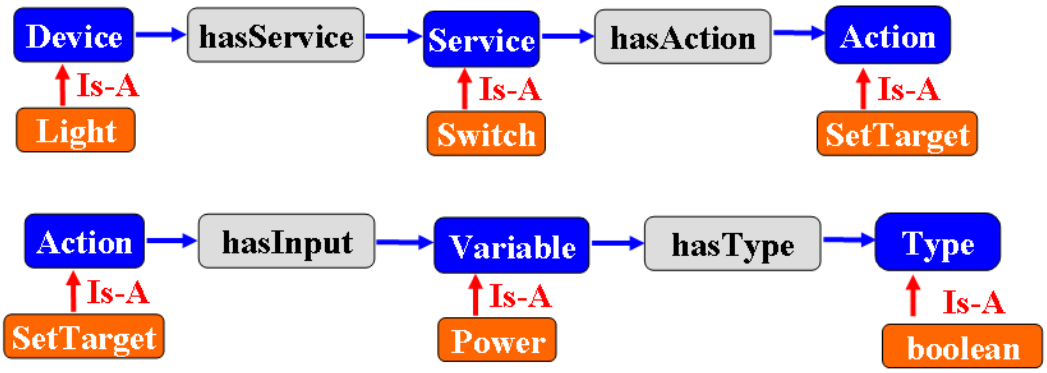


Figure 5.2: An Ontology Example

An ontology then, captures a simplified view of a certain domain through predefined important concepts. Moreover, the ontology provides a classification in a some sort of a taxonomy along with the relations between the concepts (hierarchies and constraints). The aim of an ontology is to capture the semantics of a domain through the concepts and the relation between them independently from the syntax.

Ontologies are expressed in triplets (Object - Attribute - Value). Figure 5.2 shows a part of a device ontology. The domain is the Plug and Play devices and the concepts of the domain in this example are `Device`, `Service` and `Action`. The relations between the concepts are `hasService`, `hasAction` and `Is-A`. The `Light`, `Switch` and `SetTarget` are instances of these ontology concepts. A `Light` is a device with a service `Switch` having an action `SetTarget`, see Figure 5.2. Thus, the ontology represents the knowledge semantically through relations connecting the main concepts of a domain in a machine interpretable form .

Another example is shown in Figure 5.3 which is proposed in [Pierson 09] to represent a context. The Device concept is a general concept in this ontology. The other devices like a TVSet, a Sensor or a ClimStation are represented as sub-concepts of the device concept. The location concept is used to represent information of places such as indoor/outdoor, a general type of room or a hotel room. The physical property is used to represent environmental properties such as temperature, brightness of moisture. These concepts are connected together through properties. For instance a Sensor concept informs a Physical property like the light intensity. This information about intensity is related to a location. Instances are also modeled in this ontology, the HotelRoom number 1345 is an instance of the hotel room concept which belongs to the location concept.

This ontology shown in Figure 5.3 allows to represent various elements in an environment and most importantly the relations between. In the appendix B.1, Listing B.1 shows how this ontology is represented in an XML based ontology language OWL which is presented in section 5.1.3.1.

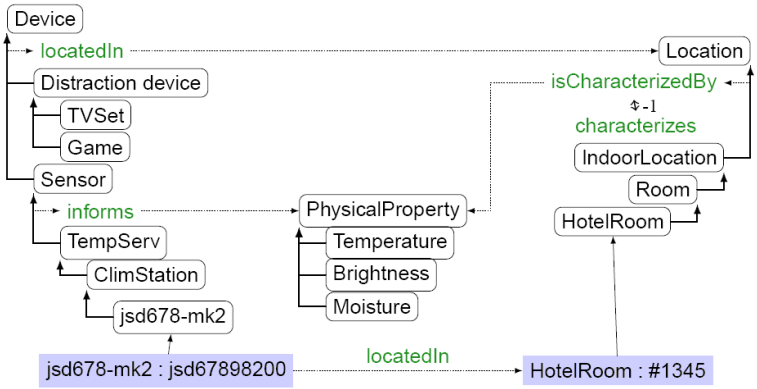


Figure 5.3: A Context Representation Ontology Example [Pierson 09]

5.1.1 Ontology Entities

Even though a lot of ontology languages have been proposed in the literature, they all deal with the following entities:

Concepts are the main entities of an ontology, they are the substitute of the things from the real world.

Individuals are considered as instantiated entities of concepts. In Figure 5.3, a `Hotel Room:#1345` is an instance if the `HotelRoom` concept.

Relations constitute the glue associating concepts between them and individuals with their concepts. In Figure 5.2, the `Is-A` relation links `Light` to `Device`, while `hasService` links the two concepts `Device` and `Service`.

Data Types are specific entities to specify a data type, such as a URI, a String, an integer or a boolean.

Data Values are values that a concept or an individual can have.

5.1.2 Ontology Development Methodologies

According to [Noy 01], the ontology development process starts by determining the domain and the scope of the ontology. Once the domain and scope are identified, the authors suggest to look up for existing ontologies. It is easier to reuse ontologies instead of building a new ones. The building process starts by enumerating important terms of the ontology in order to define the concepts and the hierarchy between such concepts. Then,

the relations between the concepts are established. Defining the data types and values then follows. Finally, only the first iteration ends by creating instances.

Multiple iterations need to be carried out to refine the ontology and adapt conflicting issues between entities in order to achieve a minimum acceptable version.

Several methodologies has been reported in the literature like in [Noy 01], [Devedžić 02] along with surveys such in [Staab 09], [Studer 07] (Chapter 3) and [Gašević 06] (Chapter 2). They all commonly agree on the fundamental rules pointed out by [Noy 01] which are the following:

- *"There is no one correct way to model a domain"*, the best solution depends on the anticipation of the ontology application and use.
- *"Ontology development is necessarily an iterative process"*, it requires to consider conflicting concepts and relations. However, after a time-consuming iterations, a minimum consensus about the ontology should be found [Gašević 06].
- *"Concepts in the ontology should be close to objects (physical or logical) and relationships in your domain of interest"*. The ontology should represent the real world, therefore the concepts and the relations selection should be close to the real world objects as much as possible.

5.1.3 Semantic Web Services

As mentioned in chapters 3 and 4, the service description exposes information about the interface and the means of interaction between services. However, the interaction between services is dependent on the correct interpretation of the representation and the associated data content as well.

Thus, semantic web services according to *McIlraith et al* [McIlraith 03], is the expressivity augmentation of web services by adding semantics to their description. The semantic augmentation is achieved by adding annotations to the service description through ontologies which represents the knowledge in a machine interpretable form. Thus, reinforcing the description of web services allows to achieve a greater level of service discovery, composition and interaction.

5.1.3.1 Ontology Languages

Various ontology languages have been designed and proposed for standardization in the literature to semantically express a domain. We outline in the following the most relevant ontology languages based on XML:

RDF(S) The Resource Description Framework [W3C 99] is based on XML and XML Schema. It has been proposed to add semantical annotation on top of XML. RDF is expressed using a triplet (Object-Attribute-Value) : The **object** refers to an identifiable web resource, like a web page, a title or an author. The **attribute** represents a property linking an **object** to its **value** which can be a text or a resource. For instance, the author *Bob* of a web page *www.abc.com/MyPage.html* can be expressed in RDF [W3C 99] as shown in Listing 5.1.

```
1 <rdf:Description rdf:about="http://www.abc.com/myPage.html">
2 <btr:hasAuthor rdf:resource="x:Bob"/>
3 </rdf:Description>
```

Listing 5.1: RDF Example

The RDF-Schema is an extension of RDF, it offers more expressivity, for example it allows to specify properties and associate them with classes. This enables to create taxonomies through reserved terms in the RDF(S) language such as *rdfs:Class*, *rdfs:SubClassOf*, *rdfs:Property* and *rdfs:Subproperty*.

RDF(S) has a limited expressivity, for instance, one cannot specify in RDFS that a class is equivalent to another class.

OWL The Web Ontology Language [Bechhofer 04] is a standard promoted by the W3C. OWL is the successor of two previous ontology languages DAML and OIL. OWL is based on the RDFS language and adds more expressivity by adding additional relations between entities. For instance, OWL allows to specify a symmetric, inverse or transitive properties between relations. It also allows to express that two entities are equivalent *owl:equivalentProperty* or different from each others through a well defined vocabulary. The (minimum, maximum) cardinality (*owl:cardinality*) of a property can also be specified. An entity can be also specified as an intersection or union of two or more other entities.

5.1.3.2 Semantic Service Description Languages

The ontologies are also used to capture semantics in service descriptions. We overview in the following semantic service description languages and frameworks built on top of the ontology languages.

OWL-S OWL-S [Martin 04] is a Semantic Markup for Web Services. It relies on the OWL language and is used to describe web services. OWL-S specifies a service with three different parts: the service profile, the process model and the service grounding.

The **service profile** provides a description of the service capabilities along with its input, output, preconditions and effects. It also allows to specify the service type and category. The **process model** describes how the service works and provides information about the service whether it is an *Atomic Process* executing alone or a *Composite Process* combining other processes. The **service grounding** holds the technical details on how to interact with the service, such as the communication protocol or the serialization techniques.

OWL-S allows composition and cooperation between web services using well defined terms to express the service entities and relations through the different profiles. However, service providers may use different ontologies to represent the input and output types, the service types and categories. Therefore, the interoperability between different ontologies is still needed.

WSDL-S The Web Service Semantics Language [Akkiraju 05] aims to extend the Web Service Description Language with semantic annotations added to the WSDL tags. It uses similar aspects from OWL-S such as precondition and effects. The WSDL-S benefits from the adoption of the WSDL as an industrial standard used to describe web services. The annotation can refer to concepts from different existing ontologies. Thus, this aspect makes the interpretation of the annotations difficult since no common semantics or relations are defined between the different ontologies and concepts. Therefore, it is almost impossible to match service requests and descriptions annotated from different ontologies using different concepts with a sort of mapping between the several used ontologies.

WSMO The Web Service Modeling Ontology[Roman 05] is proposed to describe various aspects related to Semantic Web Services. WSMO has four main concepts to specify service descriptions: Ontologies, Services, Goals and Mediators. The **ontologies** provide defined terminologies to capture the semantics of a domain. **Services** request or announce their description using the same terminology from a specific ontology. The **Goals** represent the user's view and expectation of a web service functionality and execution. And finally, **Mediators** are used to resolve incompatibilities between data and terminologies from different ontologies. Such mediators can use ontology alignment results to resolve incompatibilities. Ontology alignment techniques are detailed in chapter 8.

Other similar languages and frameworks were also proposed in the literature to add semantics to the service description. For example, the Semantic Web Services Framework (SWSF) [Battle 05] has also been proposed along with its two parts: the Semantic Web Services Language (SWSL) and the Semantic Web Services Ontology (SWSO).

The ontologies allow to capture information of a domain and to semantically express the relations between its entities. Such captured information is exploited to extract new information from what is already represented in order for example, to take a decision or interpret information. To extract information from what is already represented in the ontologies, rules combined with logics can be employed. We outline in the next section how rules can be used to infer new information from what is already represented.

5.2 Rules

Rules are another form of knowledge representation [Davis 93], [Gašević 06], [Studer 07]. Actually, human knowledge is also described using rules to make actions or think about situations. The rule structure is a combination of two parts, the preconditions and the conclusion. It is usually expressed in the form of an *IF.. THEN ...* For instance, IF "the service description matches the request" THEN "Bind to the service and interact". Rules are usually formalized using logics to provide defined semantics in order to remove ambiguity. Combined with logic, the rules are used to exploit the information from what is already available [Horrocks 03]. Even more, rules are used to infer and deduce new information from what is already represented. A typical example of the deductive inference is the famous quote of *Aristotle*: *"All men are mortal. Socrates is a man. Therefore, Socrates is mortal"*. Thus, the rules are not limited to representing knowledge but are also used to manage and extract information from what is already represented. Additionally, extracting information from existing data allows to trigger various operations which perform data classification or annotation on the existing information. Thus, the rules are used to represent and manage knowledge.

5.2.1 Logic-Based Representation

Logic based representation paradigms provide defined semantics to avoid ambiguous expressions. Such different representations allow a different level of expressivity. We overview in the following three logic based representations along with the limitations of their expressivity.

5.2.1.1 Propositional Logic

Propositional logic is about symbolic reasoning through propositions. A proposition is a logical statement assigned to a symbolic variable. A statement is written in a natural language and it cannot be interpreted by a machine. For example, "A= Device D provides Service X", "A" is a symbolic variable, it has a true or false value representing the truth of the logical statement.

Additionally, a proposition can be linked with another using logical operators such as AND($\wedge$), OR($\vee$), NOT($\neg$), IMPLIES($\Rightarrow$) or EQUIVALENCE($\Leftrightarrow$) to form more complex rules. For example, let "B= Application P requests service X" and "C= An application interacts with a device". The relations between the three propositions can be expressed as follows: $A \wedge B \Rightarrow C$. Meaning, if A and B are true then C is true.

The propositional logic allows reasoning with rules through logical operators and variables. It mainly depends on the truth tables of logical operators. The semantics of symbols is unimportant since the reasoning depend on the values of the propositions (true, false). Moreover, the truth value of a statement can be subjective and not a common truth.

5.2.1.2 First Order Logic

The first order logic extends the propositional logic and offers more expressivity by introducing two quantifiers: the Universal $\forall$ and the Existential $\exists$. It also uses constants, variables and predicates. A constant begin with a lowercase letter while a variable with an uppercase. A predicate links two constants or variables together. For example "Device D provides Service X" would be written *provides*(D, X). For example, the rule "Device D provides a service X" and "Application A requests a service X" then "Application A interacts with the device D" would be written as: $\forall$ D *provides*(D, serviceX) $\wedge$ $\forall$ A *requests*(A, serviceX) $\Rightarrow$ interacts(A, D).

The first order logic has enough expressivity and allows other representations to be based on it. However, the First Order Logic cannot represent inclusions, union or intersection relation between elements. We present next the description logic which offers more expressivity.

5.2.1.3 Description Logic

The description logic is based on the First Order logic and offers more expressivity. It allows to define a vocabulary of a domain in a knowledge base called *TBox*. The vocabulary include an atomic terminology which consists of concepts and roles. Concepts represents a set of individuals while the roles are binary relations between the concepts. From the atomic terminology, a composite terminology can be composed using various symbols such as intersection $\cap$, union $\cup$, or inclusion $\subseteq$ between elements. The *TBOX* expresses the semantic relation between entities. For example, "Hardware" and "Software" are atomic concepts. From these atomic concepts, we can propose composite concepts. For instance, "Device $\equiv$ Software $\cap$ Hardware". The Assertion Box *ABox* represents new instances using the *TBox* terminology. For example: Device(Printer) allows to specify that a Printer is a Device concept.

OWL allows to express the concepts of an ontology in description logic using its sub-language, the OWL-Description Logic language. The description logic encounters limitations when defining predicates of arbitrary arity (more than two) and in the results decidability.

5.2.2 Rule Languages

Various rule languages have been proposed in the literature such as Lisp [Steele 90] and Prolog [Clocksin 03]. We outline in the following only rule languages specified to reason and extract information from ontologies.

SPARQL This language [Prud'hommeaux 04] is specified to query ontologies based on the RDF language. A SPARQL query targets a pattern in an RDF ontology. The pattern can be a simple RDF triplet (O-A-V) or a complex one combined of multiple simple and complex RDF triplets. The SPARQL syntax is SQL-like. The SELECT clause holds variables starting with "?", the clause WHERE contains the pattern to be matched. A query of the Listing 5.1 can be specified as shown in Listing 5.2.

```
1 SELECT ?author
2 WHERE {http://www.abc.com/myPage.html btr:hasAuthor ?author}
```

Listing 5.2: SPARQL Query Example

SWRL The Semantic Web Rule Language [Horrocks 04] targets OWL ontologies and supports description logics. SWRL Rules can be based on the OWL entities. For example, the Atoms of the *TBox* can be represented using OWL classes C(x), or properties from the ontology P(x,y).

OPPL The Ontology Pre-Processor Language [Šváb Zamazal 10] was initially proposed to manipulate ontologies OWL. The OPPL allows to detect patterns and apply transformations on the ontology by creating new elements or removing others. Moreover, the semantics of OPPL are close to OWL-DL. An OPPL

statement is constituted of three parts: the variable definition, the selection and finally the actions. The variable definition part holds before the SELECT, it allows to define the variable type and category in the OWL language. The selection part allows to query the ontology searching for the specified pattern. The action part allows to apply transformation on the ontology by adding or removing elements and properties between. Listing 5.3 shows an OPPL statement example which links a device and an application providing/requesting equivalent service types in the ontology.

```
1 ?device:CLASS , ?serviceX:CLASS , ?application:CLASS , ?serviceY:CLASS
2 SELECT ?device subClassOf providesService some ?serviceX ,
3 ?application subClassOf requestsService some ?serviceY
4 WHERE ?serviceX equivalentTo ?serviceY
5 BEGIN ADD ?device subClassOf canBindWith some ?application END;
```

Listing 5.3: OPPL Query Example

Ontologies allow to semantically represent a domain while the rules combined with logics applied on ontologies enable to infer and extract information from what is already represented.

Another interesting knowledge representation vision is tackled by the "Models" which offer different layers of representation. The high level layers manipulate abstract concepts while the lower level layers contain concepts more close to the real objects with implementation and technical related details. Thus, the high level layers offer to the designers the ability to manipulate abstract concepts with a minimum specific and technical information. Moreover, the "Models" domain offers the ability to generate low level concepts from the high abstract ones. Thus, compared with the ontologies, the "Models" offer a knowledge representation with abstract concepts away from detailed information dependent on the technical details of the real world objects. We outline next the knowledge representation using "Models".

5.3 Models

Hagget et al [Hagget 67] define a model as a simplified abstraction of reality. An abstraction is also considered as another form of knowledge representation. Figure 5.4 shows different levels of abstraction. The M0 layer represents objects from the real world, such as a real printer. At the M1 layer, a drawing of a printer abstracts the printer from the real world. Thus, the printer's drawing is a model abstracting an object from the real world. The M2 layer, the meta-model layer specifies the basic concepts to be used by the underneath layer. Thus, each model (i.e. drawing) at the M1 layer is constituted only of concepts from the meta-model such as a circle or a triangle. Therefore, models (i.e. drawings) at the M1 layer are conformed to the M2 layer since they are constituted only from the specified concepts at the M2 layer. The M3 layer also defines concepts to be used by the M2 layer models and so on. However, the last layer Mn also includes self-defined concepts to be used by the layers underneath and by the Mn layer itself. In this example, the dot concept is the self defined concept which allows to define all the concepts at the lower layers.

A layered model architecture is proposed by the Model Driven Architecture (MDA) [OMG 03] which is promoted by the Object Management Group (OMG)[1]. The MDA defines standard models and languages like the Unified Modeling Language (UML) [OMG 97] to specify concepts at the M1, M2 and the Meta Object Facility (MOF) [OMG 06] to specify concepts at the M3 layer. Additionally, relations between concepts along with the allowed cardinality is specified by each upper layer. The MDA provides a four layer architectural modeling space: M0 as the instance layer, M1 for the model, M2 for the meta-model and the M3 for meta-meta-model layer.

[1] www.omg.org/mda

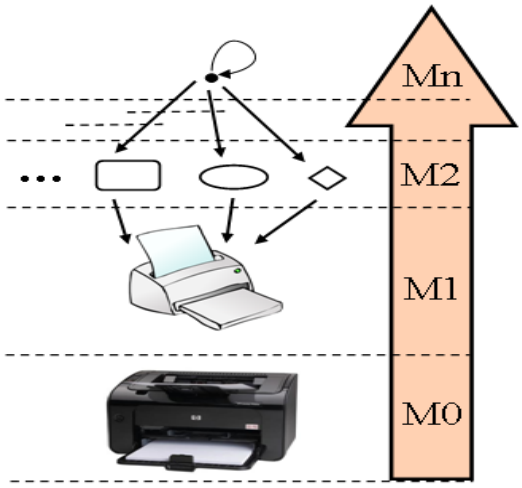


Figure 5.4: Model Driven Architecture, Generalized Layers Example

The Model Driven Architecture is considered as a specific instantiation [Bézivin 05] of the Model Driven Engineering. The MDE [Schmidt 06] is a software development methodology based on different levels of abstraction aiming to increase automation in program development. The basic idea is to abstract a domain with a high level model then to transform it into a lower level model until the model can be made executable using rules and transformation languages like in template-based code generation tools. The MDE is used to improve software development through automatic code generation from higher to lower layers.

In the pervasive applications development, a lot of works have been proposed in the literature which provide a high level abstraction model to specify applications. Such specification is carried out by designers through high level languages detailing the concepts to be used, their behavior along with the interaction between. Then, using MDE techniques, the high level language is automatically transformed into an executable code. Thus, high level specification allows designers to focus only on the high level aspects in the application development which are mainly the design of modules, their behavior and the interactions between. For instance, designers will only manipulate high level modules such as a light device concept, a clock concept and a movement detector concept. The designer should only be aware of the concepts functionalities and their behavior without knowing the technical details allowing applications to communicate. Additionally, the designer will specify the behavior of each module and how it should react upon receiving events from other modules. Once the specification is achieved, code generators transform the high level specifications into lower levels.

Identically, the development process of applications goes through almost the same levels: the specification, then the implementation which holds a lot of technical details such as how to interact with real devices or what data structure should be sent. Thus, automating the transformation from high level specifications into low level code generation can actually save a lot of time for designers and can reduce the human errors occurring during the programming phase. The automation is carried out by transformation rules previously written by designers.

Moreover, the high level concepts help designers and developers to share and to agree on a common representation of the system to be built.

5.4 Conclusion

In this chapter, we presented different techniques to represent information. Ontologies are used to capture the semantics of a domain. Each important element of a domain is represented as a concept then various properties

relate concepts together. Individuals can also be represented and are considered as an instance of a concept. Data types and values are also represented in ontologies. Ontologies can be used in services description to provide a common semantics which can be shared between service providers and consumers. Different languages has been proposed to increase the expressivity of the service description. Such expressivity is supported through semantic annotations of the service description.

We also presented rules as a mean to exploit information from ontologies. Combined with logics, rules offer defined semantics in order to remove ambiguity. Additionally, rules represent a powerful mechanism to reason and extract new information from what is already represented. Different rule languages are proposed to query ontologies or to detect predefined patterns and apply transformation and modifications on a ontology.

Models also help to capture knowledge on different levels of abstraction. The Model Driven Architecture specified a four layer architecture to abstract information and systems. Additionally, the MDA allows the specification of concepts manipulated at a certain layer only from concepts defined at an upper layer. This allows to express the concepts in a well defined and precise semantics. Combined with the Model Driven Engineering techniques, models offer an automation of software components creation based on high level descriptions through code generation techniques.

In our contribution, we present how these three representation techniques: "Ontology", "Rules" and "Modles" combined together can be used to resolve the device and service heterogeneity and achieve interoperability.

Chapter 6

Conclusion & Problem Statement: Device Interoperability

"A problem is a chance for you to do your best".

– Duke Ellington

In the part I of this thesis, we outlined the context of ubiquitous computing. We drew in chapter 2, the global vision of a ubiquitous environment than we extracted its characteristics. Based on such characteristics, we summarized the important features a ubiquitous system should support.

From an ideal vision of a ubiquitous environment and system, we presented the digital home, a current environment moving towards a ubiquitous one. We detailed the actual state of the digital home along with the current available technologies. Then, we pointed out the following three challenges in such an environment. First, the **dynamic discovery** of existing devices along with their supported services and capabilities. Second, the **interoperability** between various plug and play devices and services. In an ideal ubiquitous environment, applications interact with devices and other applications transparently to accomplish a specific task such as printing or rendering a video. Thus, applications should be set free from a device or protocol technology. And finally, **management operations** in the digital home are needed mainly to control, deploy, troubleshoot and diagnostic different devices and applications. However, the heterogeneity of the plug and play protocols adds a burden cost on the telecoms operators, device manufacturers and applications providers. Therefore, those three actors must maintain a wide variety of administration and diagnostic tools in order to support troubleshooting and maintenance on different devices communicating through heterogeneous protocols.

Obviously, these three challenges pointed out in the digital home environment must be tackled by a ubiquitous system architecture capable of supporting the following features: the **discovery**, the **control**, the **eventing** and the **interoperability**.

Therefore, in chapter 3, we exposed the Service Oriented Architecture and showed how the SOA characteristics meet at best so far the ubiquitous system characteristics. In fact, the SOA architecture consists of three elements: service providers, consumers and the service registry. Service providers publish their service description which allows service consumers to **discover** the provided capabilities in order to **control** the service providers. Additionally, the SOA architecture supports the **eventing** between services. In fact, Plug and Play devices in the ubiquitous environment announce their description while ubiquitous applications listen to such advertisements in order to interact with specific devices to accomplish a given task.

Thus, the SOA can provide to a ubiquitous system the following characteristics: discovery, eventing and

control. Moreover, the SOA refines the plug and play device heterogeneity into a SOA service description heterogeneity. In fact, the main limitation of the SOA resides in the service description which is expressed only syntactically, thus making the service matching dependent on the syntax description instead of the semantics.

In order to identify more deeply the plug and play device heterogeneity, we detailed in chapter 4 the following plug and play protocols: UPnP, DPWS, Bonjour and IGRS. Devices supporting such protocols follow the service oriented architecture.

We also drew a detailed plug and play protocols comparison showing their commonalities and divergence areas. Figure 6.1, points out the three characteristics involved in achieving a complete plug and play device interoperability: Discovery, Control and Eventing. The three characteristics cover the L1, L2 and L3 layers mentioned in chapter 2 where heterogeneity has settled in the protocols stacks, the device and service description format, and the description content. We overview next, the heterogeneity in each layer.

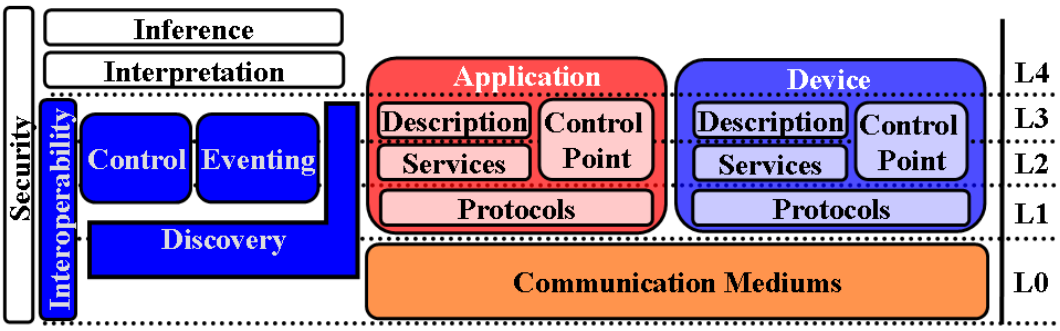


Figure 6.1: Plug And Play Interoperability Layers

Protocol Stacks Each protocol actually supports its own stack containing multiple protocols to allows discovery, eventing and interaction. Thus applications searching for various devices need to support the different protocol stacks in order to discover and interact with different protocol devices.

OSGi and the base drivers allow to represent devices and their services as local OSGi services which can be invoked independently from the protocol stack. Base drivers will transfer local invocations received by the OSGi reified services to real devices. The OSGi and its base drivers move then the protocol stack heterogeneity into the base drivers API calls. However, the service description and the content heterogeneity remain to be solved.

Device and Service Description Each protocol also defines a specific description format to announce the device information and the services description. UPnP uses an XML template based description, while IGRS and DPWS uses the Web Service Description Language. Bonjour also defines its description format. Thus, the heterogeneity in the description representation makes it impossible to applications or devices to interpret the announced description or the received events and interact with the provided service.

Description Content Another layer of heterogeneity is present on top of the description content. Each protocol defines a set of standard profiles per device type which imposes standard devices to use the same content syntax to describe the supported services, actions and parameter types and names.

Since each protocol defines its own standard content, the interoperability is impossible. The discovery will not occur since an application searching for a UPnP Printer with a type "PrinterDevice" only binds to device holding the same specific name. Thus, a DPWS printer holding the name "PrintingDevice" will not match even though the application is searching for any device capable to print.

The interaction and control cannot occur since each action is described using various content. For instance, a standard UPnP printer supports the action `CreateURIJob` to print a document while a DPWS standard printer supports an equivalent action expressed syntactically differently, the `CreatePrintJob` action to print a document. Moreover, since applications and devices interpret the description syntactically, then an application cannot interact with a UPnP printer since the content description is different, even though the printer supports semantically equivalent actions.

The three levels heterogeneity enclose smart applications into specific and preselected devices and services orchestration. Smart applications need to be set free from protocol and service syntax heterogeneity. The user must not be restrained to one type of protocol and devices, but should be able to integrate easily and transparently equivalent devices to his home environment.

The description and the content heterogeneity along with the protocols layers diversity, prevent applications to discover and use any available equivalent device on the network to accomplish a specific task. Designing ubiquitous applications to support multiple protocols is time consuming since developers must implement the interaction with each device profile and its own data description. Additionally, the deployed application must use multiple protocols stacks to interact with the devices. An interoperability between plug and play devices requires an interoperability on the three different layers.

Moreover, application vendors and telecoms operators need to orchestrate devices through a common application layer [Spets 10], independently from the protocol layers and the device description. Figure 6.2 overviews the application vendors and telecoms operators vision to interact with plug and play devices. They would like to develop applications interacting with plug and play devices transparently through a common application layer [Spets 10]. Such common layer actually abstracts the three layers of heterogeneity and allows ubiquitous applications to interact with any device through common instructions specific to a device type. For instance, the UPnP and DPWS Light devices would be controlled using the same actions provided by the common application layer. It is up to the common application layer to adapt and translate the commands according to the specific target device using its own protocol stack, description and content.

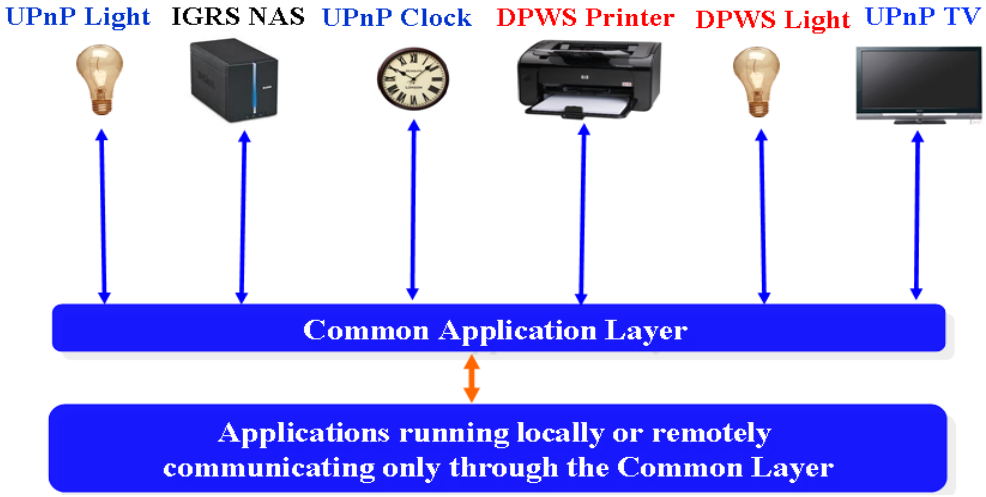


Figure 6.2: A Common Application Layer

However, proposing a common application layer is very complex since the common layer will need to capture common concept and find common semantics between the actions from the same type device. Additionally, common parameters (types and content) need to be proposed which is not simple since some actions require a set of obligatory elements and other optional ones. Therefore, the main focus of our work will concentrate on

the service adaptation to allow a plug and play interoperability between devices and their provided services.

To address the description representation format and content heterogeneity, we need to clarify through actual available techniques, how knowledge and information can be represented. Therefore, we provided in chapter 5 three knowledge representation techniques: ontologies, rules and models. Ontologies are used to capture semantic relations between important concepts of a domain. Rules combined with logics offer formal semantics to extract and reason on information from what is already represented. And finally, models allow to capture information with different details. Models also offer an automation of the software components creation based on high level descriptions through code generation techniques.

The rest of the document is organized as follows: the next part overviews related work in the literature proposing different solutions to hide the service description heterogeneity and allow device interoperability through service adaptation. Then part III details our contribution and shows how a combination of the three representation techniques can be used to resolve the plug and play device heterogeneity.

Part II

Related Work

Chapter 7

Overview of the Interoperability Frameworks

"Because of the lack of interoperability, we can lose billions of dollars in productivity."

– Charles Giancarlo

Contents

7.1 Common Ontology **92**

7.2 Abstract Model **95**

7.3 Uniform Language/Interface **98**

7.4 Comparison & Discussion **101**

We survey in this chapter, research efforts proposed in the literature to allow service adaptation. We refer to the term adaptation as a transformation of a service in order to adjust to precise conditions. Such conditions as mentioned in chapter 2, are due to various reasons such as a device appearance, user demand, scarce resources, and many others.

The service adaptation can occur at the compile time or at run-time, on demand or dynamically triggered.

A service adaptation is involved in one or multiple aspects. For example, it can include a transformation in one of the following: the input/output data of a service, the service interface, the communication medium, the service behavior, the resource allocation and use, platform migration and execution and many others.

We focus in this chapter only on the service interface adaptation, input/output data and behavior. Different approaches have been developed to solve the interoperation problem through service adaptation, it can be put in three major categories. **The first category** uses a **common ontology** capturing all the semantics of a domain. The adaptation process queries the common ontology which holds all the concepts from a domain and the relations between them. The adaptation is possible only if the service description uses the same concepts of the ontology. The adaptation is usually performed at run-time.

The second category is based on a **abstract model** capturing abstract and high level concepts of a domain. Then, using transformation rules and code generation techniques, an adaptation at the lower level is achieved. The adaptation can be performed at run-time or at compile-time.

The third category handles the adaptation through a rewriting of the service description using a **universal language** or a unified interface. The adaptation is usually not performed at run-time since the description is re-written offline.

7.1 Common Ontology

This section overviews research work proposing service adaptation for interoperability based on a common ontology. The global ontology is manually built by an expert and represents the main concepts of the domain and the relations between such concepts. A common ontology is used mainly to classify concepts in a hierarchical organization which is referred to as taxonomy. Figure 7.1, shows an example of a common ontology. The `Thing` concept is the root element in an ontology. The `Printer` concept for example is a sub-concept of the `Device` concept.

For conciseness, we choose to detail only three service composition approaches. Paolucci's semantic matching algorithm, the PERSE and the MySIM middlewares.

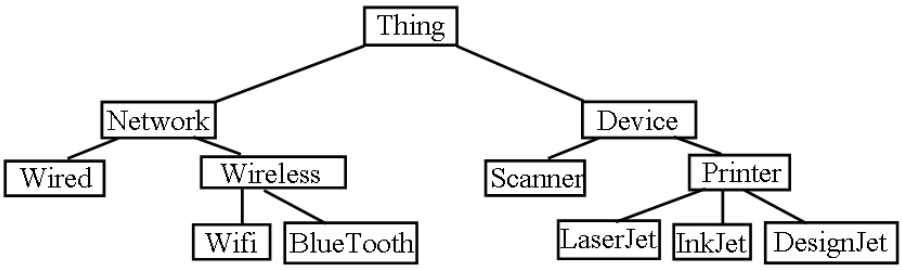


Figure 7.1: A Common Ontology Example

7.1.1 Paolucci's Semantic Matching Algorithm

As mentioned in chapters 3 and 4, the service registry satisfies service requests by applying a syntactical matcher. Thus, service consumers and service providers must use the same syntax when publishing or searching for a service. *Paolucci et al* propose in [Paolucci 02], a semantic matcher algorithm to extend the syntactical matching in a service registry. The algorithm takes as an input a service advertisement(A) and a service request(R), then reasons on their input and output parameters based on their semantic relations. The input and output parameters description uses only the concepts from the common ontology. *Paolucci* defines four degrees of matching (see Listing 7.1) to reason on the service matching.

```
1 degreeOfMatch(out R, out A):
2 if outA = outR then return exact
3 if outR subclassOf outA then return exact
4 if outA subsumes outR then return plugIn
5 if outR subsumes outA then return subsumes
6 otherwise fail
```

Listing 7.1: Paolucci's four matching degrees

The algorithm differentiates between four degrees of matching:

Exact matching occurs if the requested output and the advertised output use the same exact concept, line [2] of Listing 7.1. An exact matching also occurs if a sub-concept is used, line [3] of Listing 7.1. For instance, if the output R is a subclassOf output A, i.e. if the service provider advertises an output parameter having as a type `Device` and the service consumer requests an output parameter having as a type `Printer`, then there is an exact match between the two outputs.

PlugIn occurs if the advertised output subsumes the requested output. For example, if the `Device` is advertised and `LaserJet` is requested. Since `Device` is a more general concept than `LaserJet`, the provided output

can be a `LaserJet`. There is a weak relation between the two concepts.

Subsume if the requested output subsumes the advertised output. For instance if the requested output is a `Device` and the advertised output is a `LaserJet`. Modifications on the service consumer are probably needed.

Fail if their no relation of subsumption between the the advertised and requested outputs. For example, there is no relation in the ontology of Figure 7.1 between `Scanner` and `LaserJet` or `Wifi`.

The algorithm returns to the service consumer, a reference to bind with the service provider. The exact matches are returned first, then the *plugIn* and finally the *Subsume*.

This approach is relatively close to the inheritance principle in the object oriented programming where the adaptation is resolved by casting objects. However, *Paolucci* defined rules to limit the substitution by restricting the inheritance degree. The more the gap between two concepts increases, the less the substitution between concepts is possible.

Paloucci implemented his algorithm and applied tests using a UDDI [OASIS 04] service registry and used the DAML-S language to annotate the service description.

7.1.2 PERSE: PERvasive SEmantic-aware Middleware

Mokhtar [Mokhtar 07] proposes a semantic middleware which provides a semantic service matching, registration and composition. In [Mokhtar 06], the semantic discovery and matching is based on the service description annotation with concepts from one common or various ontologies. In [Mokhtar 06], they proposed a semantic matching algorithm based on the semantic distance between concepts in the ontology, i.e. the levels separating two concepts in a common ontology. In [Mokhtar 08], they proposed a service matching based on Paolucci's semantic algorithm. They also take into account the service non-functional properties such as the QoS during the service matching.

An evaluation of the semantic service matching showed that most of the time is spent during reasoning and concepts classification. Therefore, they proposed an encoding mechanism to represent the service description (input, output and category) with regard to the ontology concept. Their encoding approach revealed to be efficient since the classification and matching time were highly reduced.

They also proposed an Interoperable Service Description Language (ISDL) [Mokhtar 07] quite similar to OWL-S, previously described in chapter 5. A translation mechanism is introduced to represent other service description in ISDL. ISDL allows to represent service description with semantic annotation and is based on the WS-BPEL [Alves 07] which allows to specify a service behavior.

The PERSE middleware also supports service composition, i.e. assemble or orchestrate many available services to satisfy users' task. The general idea is to transform each service behavior specified in the service description to a finite state automate then to select only sub-automates verifying the user's task constraints. Then, the selected sub-automate is used to satisfy the user's task by composing various services in the environment.

The PERSE [Mokhtar 07] middleware has been implemented in Java and validated using the Salutation (ex-Bonjour) and UPnP protocols, they also used a UDDI service registry.

7.1.3 MySIM

Ibrahim [Ibrahim 08] presents the MySIM middleware which allows a service matching, transformation and composition. The service description is annotated with concepts from a common ontology. MySIM takes into account during the service matching and composition, non-functional properties such as the QoS which can be specified in the service description as shown in Listing 7.2. Each service is annotated with a predefined concept

from the common ontology, such as the "printer" concept. The input and output parameters are described in triplets, <parameterName, type, semantic concept>. Additionally, each description contains non functional properties which can be qualitative *Npql* (line[4] in Listing 7.2) or quantitative *Npqn* (line[5]). The adaptation carried out by MySIM is transparent to the user and the application. The service matching is syntax and semantic based and takes non-functional properties into account.

```
1 Cpt= {<print, "printer" >}
2 In = {< f, java.io.File, "document" >}
3 Out= {<java.lang.Boolean, "state" >}
4 Npqn= {(nbPage,60,>), (price,10,<)}
5 Npql= {(access, "wifi")}
```

Listing 7.2: Printing Service Description Example in MySIM

Figure 7.2 shows the internal architecture of the MySIM middleware. It is composed mainly of 4 modules.

The *Translator* module translates OSGi services and properties into the MySIM service model (see Listing 7.2) which holds semantic service description using the OWL-S language. The OSGi service implementation already contains the semantic annotations relating to concepts from the common ontology. The annotation are specified on the OSGi service properties, see section 3.3 and Listing 3.1 for an example of service properties.

Based on this translation, the *Generator* module proceeds with the semantic and syntactic matching of services and also composes services based on their functional services and descriptions. The matching is based on the Java introspection techniques. First the syntactic matching takes place, then, the semantic matching which relies on *Paolucci*'s algorithm assisted with a semantic reasoner.

The *Evaluator* module verifies that the selected services for composition or adaptation are conform to the requested or provided non-functional properties. Moreover, the *Evaluator* monitors the arrival and departure of services in order to compose new services. It requests from the *Generator* module syntactic and semantic compatible services to compose.

The *Builder* is the executor of the adaptation which applies an interface transformation or a composition of one or multiple services. The *Builder* composes a new bundle from the selected services, it either copies their implementation in the new bundle or simple implement a call to the appropriate service. The *Builder* then generates and installs the new bundle for the newly adapted service.

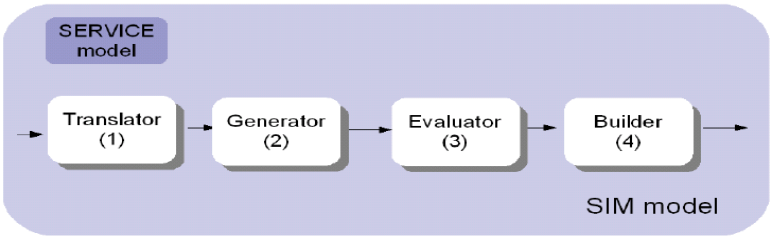


Figure 7.2: MySIM Middleware [Ibrahim 08]

Other approaches have been proposed in the literature based on the use of a common ontology. In [Redondo 08], the OWL-OSGi is proposed as an extension to OWL-S to enhance the semantic matching of OSGi services along with a common ontology to reference OSGi services. In the work of [Vallée 05], a service composition is also proposed using an annotation of services using OWL-S referring to predefined concepts. *Fujii et al* [Fujii 05] also focus on satisfying users requests through service composition. A user's request can be expressed with an "intuitive form" like a natural language, than it is transformed into a semantic request. Then, the request is

satisfied through a service composition from already available services. *Fujii*'s approach also uses annotations from what they refer to as a "Semantic Graph".

7.2 Abstract Model

Abstract models are also used in middlewares to provide interoperability between services and protocols. As shown in chapter 5, models can be used to capture information about a domain. Assisted with model driven engineering (MDE) [Schmidt 06] techniques, an abstract model can be transformed into an executable code. Thus, the methodology consists in proposing an abstract model independently from a language or a technology then to transform it into an executable code through transformation rules and code generation techniques. We present in this section, three middlewares relying on MDE techniques and abstract models.

7.2.1 DOG: Domotic OSGi Gateway

The Domestic OSGi Gateway [Bonino 08b, Bonino 08a] uses a common ontology as a taxonomy to classify devices and as an abstract model to generate specific code through transformation rules. DOG aims to provide interoperability between basically simple devices (simple light and switches). The DOG framework targets mainly simple protocols such as Konnex [Konnex 04], MyOpen and X10 [Smarthome 04]. The DogOnt ontology is inspired from the *European Home System* taxonomy which provides a classification for multiple goods in a home environment. Thus, the DogOnt captures a device classification of home equipments. Additionally, functionalities are described apart and then associated to devices in the ontology according to its category. The functionalities are separated into three different abstract categories: control, notification and query. The control functionality models the ability to control a device such as a light. The notification captures the ability of a device to inform about its internal state change. And finally, the query functionality which describes that a device can be queried to retrieve information such as its internal state values. Figure 7.3 shows two instances represented in the ontology, a dimmer lamp and an on/off switch and the relations between. The figure also exposes the complexity of representation a device in the DOG approach. An added device such as dimmer lamp must be connected with other existing actions and commands in the abstract ontology.

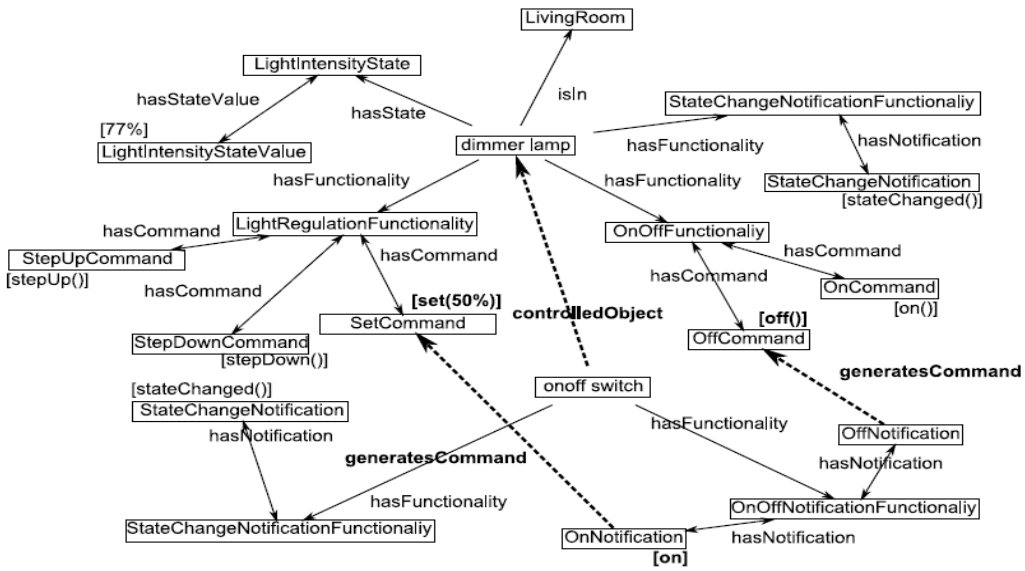


Figure 7.3: Partial DogOnt Fragment showing Dimmer Lamp and Switch Instances [Bonino 09]

Domotic designers individuals add device instances to the ontology manually, then a set of SWRL rules

(see chapter 5) are applied to connect some properties to the instance. For example, if the instance is a light device then a control functionality is automatically added. Moreover, it is up to the designer to add the device description to the ontology which contains the device name as well as all the low level details like the device address and its format (see the simpleHome[1] ontology which an instance of the DogOnt).

DOG uses specific Network Drivers similar to the base drivers described in chapter 3 to communicate with the real devices. On DOG start up, the Platform manager [Bonino 08a] queries the DogOnt and retrieves the list of device instance to be manage along with their descriptions and their low level details such as the address and the communication protocol type.

When DOG receives a command such as a turn off light command for a specific light device. Based on the device light type and address, DOG queries the ontology using SPARQL [Bonino 08b] and SWRL [Bonino 10] to retrieve the allowed parameters and constraints. Once the command passes the syntactic and semantic validation, *MVEL*[2] rules are then used to generate specific messages which are then sent to the real device.

7.2.2 EnTiMid

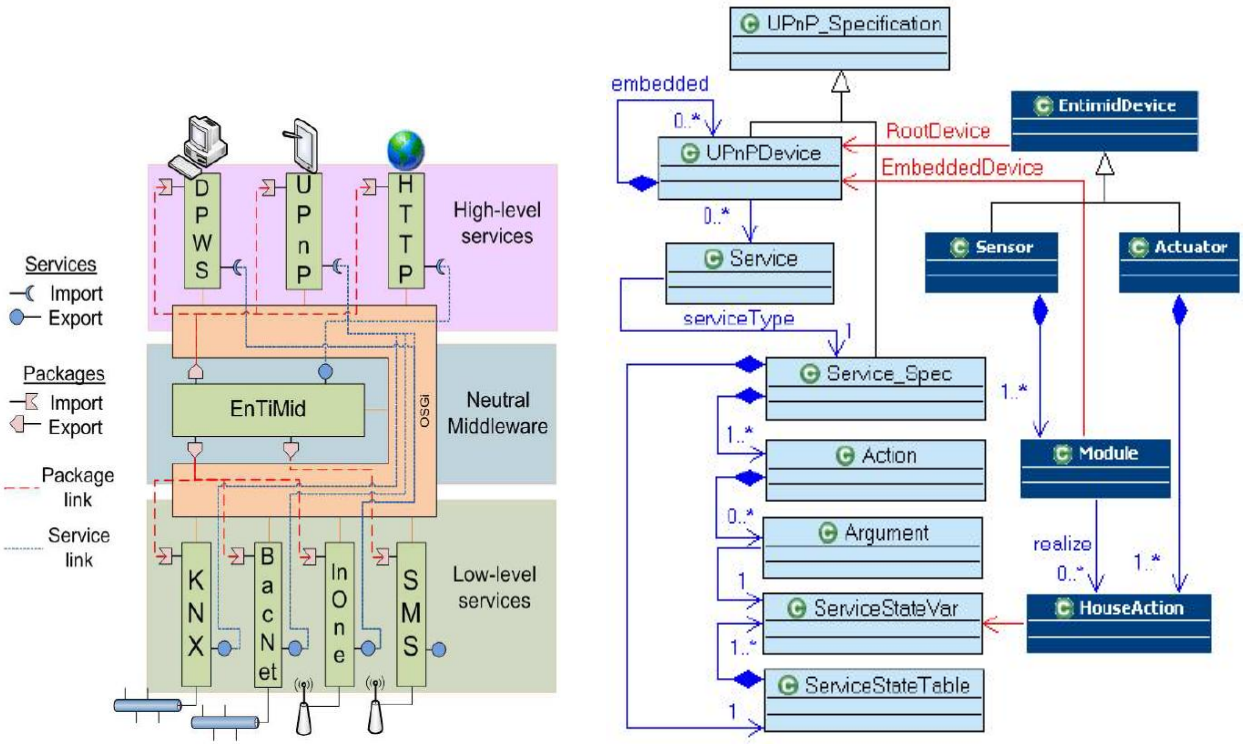


Figure 7.4: (a) EnTiMid Architecture (b) An EnTiMid-UPnP Model Mapping example [Nain 08]

The EnTiMid [Nain 08] middleware proposes to generate, using MDE techniques, service implementations based on a predefined abstract representation. The left part of Figure 7.4 shows the internal architecture of the EnTiMid middleware. The aim of EnTiMid is to represent devices supporting specific and proprietary low level protocols like InOne, Konnex or LonWorks as HTTP, UPnP or DPWS services. The low level layer holds relatively simple devices such as temperature sensors, door actuators, lights and switches.

The right part of Figure 7.4 shows the EnTiMid device model which represent a low level device by either

[1] http://elite.polito.it/ontologies/simpleHome.owl
[2] `mvel.codehaus.org`

a sensor or an actuator device. An device supports a set of *Common Actions* such as (on, off, up, down). Each common action provides specific information for the action to be executed on a specific device, instantiated by a domotic designer. A sensor device is responsible to interact with an actuator device. For example, a detection sensor can be in charge of switching on a light. During the setup, the sensor queries the light actuator to retrieve specific information on how to interact with it.

Users or domotic designers instantiate devices they need to interact with from the abstract model. The mapping between the EnTiMid model and the UPnP model allows to go from one model to another using transformation rules. Another set of rules are used to automate the code generation of a UPnP software device, to go from a high level into a lower execution level.

The EnTiMid paper [Nain 08] don't detail how the users instantiate actions on an actuator device. The authors don't provide information about the transformation rules to go from the high level model to the generated code.

7.2.3 PervML: Pervasive Modeling Language

In [Serral 08], the Pervasive Modeling Language is proposed to allow pervasive applications developers to deal with service composition and heterogeneity. The PervML is designed to easily specify and describe functionalities of pervasive devices and services in conceptual models which are technology independent.

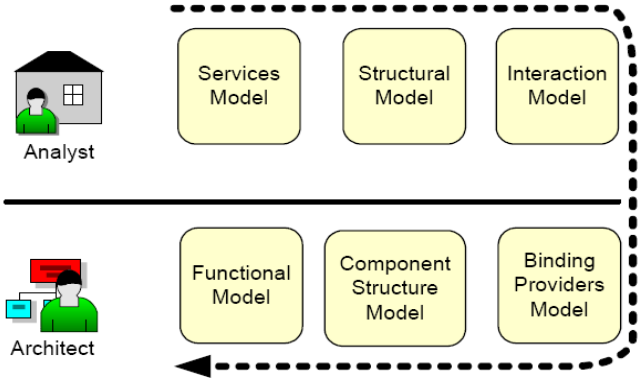


Figure 7.5: PervML Models [Munoz 04]

According to [Serral 08], developers are categorized in two roles: system analysts and system architects, as showed in Figure 7.5. Each role uses a set of 3 models to capture information using the PervGT (Pervasive Generative Tool) [Cetina 07]. A system analyst uses the *services model* (see Figure 7.5) which describes the provided services by the system along with their actions, e.g. a lighting service provides dimming actions. A *structural model* which provides information about the available instances of each service and the relations between services and their interfaces. And the *interaction model* which describes through sequence diagrams the interaction between services.

A system architect uses three models to bind the supported services to available physical devices. The *Binding Providers model* offers the different devices in the system along with their supported operations. The *Component Structure model* binds the available instances specified by the system analyst in the *structural model* to the available device instances. And finally, the *functional model* is used to specify the operations to be executed on a service invocation.

After the modeling step, the PervML models are transformed automatically through code generation techniques into an executable OSGi bundles. An OWL ontology also captures the services functionalities and other

information to allow a runtime adaptation through reasoning on situations such as the user's location or to predict his next action.

Other MDA-based works have been proposed in the literature. *Coopman et al* [Coopman 10], proposes a common ontology used as an abstract model to represent devices and their functionalities along with their interoperable relations. The ontology in their approach is extended from both the SOUPA [Chen 05] and the DogOnt [Bonino 08b]. The WComp [Tigli 09a] middleware also adopts a high level model to specify service components using an graphical user tools. From the high level specification, specific components are generated and installed in the WComp middleware to allow service composition. The Gator Tech Smart House [Helal 05, Helal 09] also provides the Device Description Language (DDL) [Chen 09] to specify device's information along with its supported capabilities. The high level description is then used to generate OSGi bundles to interact with sensors and actuators through network converters similar to the OSGi base drivers.

7.3 Uniform Language/Interface

The last category uses a uniform language or interface in order to represent devices and their supported capabilities in a universal representation format. Additionally, some of the proposed works in the literature define a universal content representations either per device type (printer, TV) or device domain (multimedia, printing). We depict in the following three approaches using a universal language or interface.

7.3.1 HomeSOA

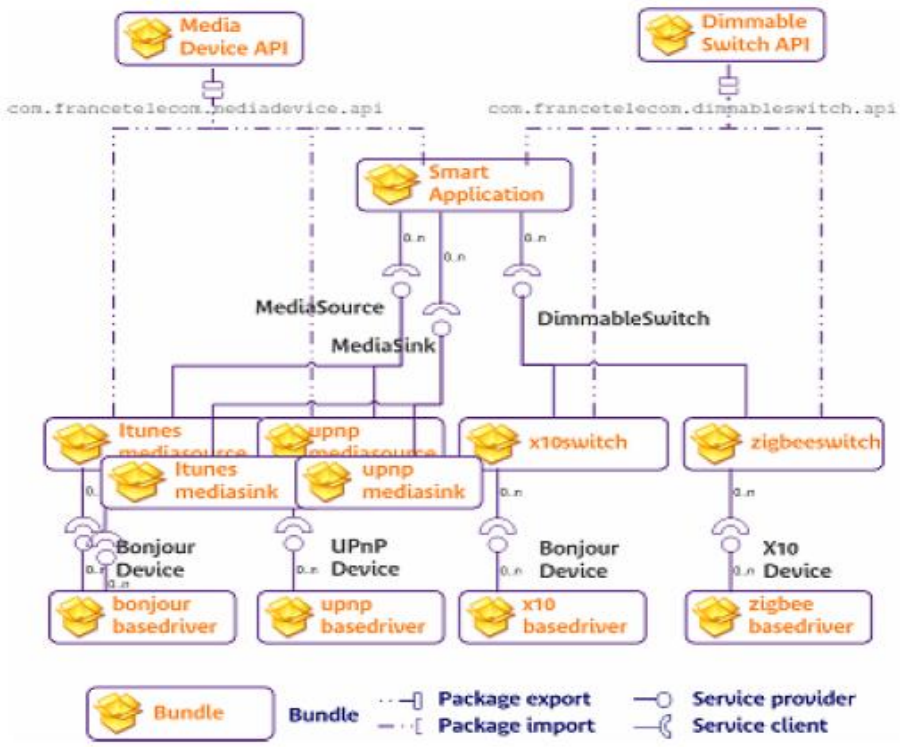


Figure 7.6: The HomeSOA Architecture [Bottaro 08a]

The HomeSOA middleware [Bottaro 08b, Bottaro 08a] adopts a uniform interface approach to represent devices semantically similar or belonging to the same domain. Figure 7.6 shows the various layers of the

approach. In the first layers reside the Plug And Play base drivers. *Bottaro* proposes a DPWS and a Bonjour Base Drivers [Bottaro 07a]. However, as we mentioned in chapter 4, Bonjour is used only for discovery, while other protocols such as DMAP for *itunes* are used on top. Thus, the Bonjour Base driver is used for Bonjour devices discovery and another layer is added to represent *itunes* devices using DMAP/DAAP. The base drivers as detailed in chapter 3, reifies real devices on the network as local OSGi services re-exposing the same device representation and content of the real devices.

Another layer of drivers, one for each protocol, called *refined drivers* at the second layer (see Figure 7.6) re-exposes predefined device types and services as smart devices and services. Each smart device exposes its original description and capabilities using a WSDL file. Additionally, a smart device uses a unified interface name for the semantically equivalent devices. For instance, an X10, UPnP and DPWS lights are reified by the first layer through the base drivers as OSGi services. Then the refined drivers re-expose the reified devices as smart devices implementing a unique interface such as the *Dimmable Switch API*.

The refined drivers re-expose devices based on their supported operations type. For instance, since blinds also can vary their aperture and closure values, then they can expose dimming operations. Thus, the refined drivers represent the blinds using the *Dimmable Switch API* which is common for blinds and light devices.

The same applies for the other device types, a UPnP refined driver *mediasource* searches for a UPnP Media Server and re-exposes it as *mediasource* using a media device interface and exposing its description and capabilities in a WSDL file.

The smart devices reification allows ubiquitous applications to search for devices by category of operations and interact with their services and actions using the Smart Device API. However, the smart API only re-exposes the device description format using the WSDL. The description content is not resolved by the smart API. In other words, applications must know in advance the description content of the device to interact with it. For instance, an application must use the action *SetTarget* to interact with a UPnP light and the *Switch* action with a DPWS light, however, it will rely on the dimmable switch API to search for the lights and retrieve their descriptions.

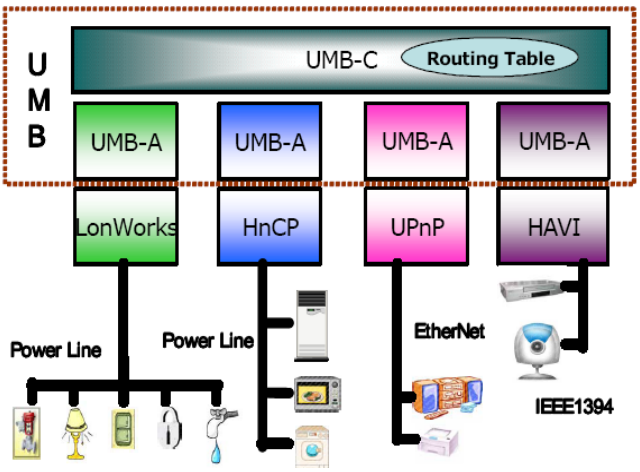
7.3.2 UMB: Universal Middleware Bridge

The Universal Middleware Bridge [Moon 05] proposes a solution to provide interoperability between various protocols: Jino, LonWorks, UPnP. *Moon et al* propose to represent each device description using a Universal Device Template (UDT) which is inspired from the UPnP representation format.

As shown in Figure 7.7, the approach uses a UMB adapter for each protocol to represent each device as a virtual proxy device on the universal middleware platform. The virtual proxy exposes the local device description (UPnP, Havi, etc) referred as Local Device Template into a Universal Device Template format. The content semantics are also translated into unified semantics. The translation is carried out by the UDT-LDT mapper which uses a mapping table containing the matching between universal and local device types, functions, parameters and events.

In the right part of Figure 7.7, a virtual proxy device mapping is showed. The table holds the mapping between the universal semantics used and the local description of each device. For instance, the table shows that the local device type *0x05 0x01* of the *LonMark* protocol is exposed as a *LAMP* in the global device type representation.

A specific router in the UMB architecture is used to route message and events to the proxies. The proxy messages are forwarded by the UMB adapters to the real devices.



Property	Field Semantic	Example
Index	DB index	0
Global/Local	Global or Local Device	Local
GID	Global ID	LonWorks, LID
LID	Local ID	0x112233445566
GDType	Global Device Type	LAMP
LDType	Local Device Type	0x05 0x01 (LonMark Object)
Local address	Local Device Address	SubnetID/NodeID
Function List	Available Function List	F1
		F2,...
Action List	Available Action List	...

Figure 7.7: UMB Architecture and a Virtual Device Proxy information [Moon 05]

7.3.3 DomoNet: Domotic Network

Miori et al. [Miori 10, Miori 06] propose the Domotic Network architecture to provide interoperability between devices supporting (UPnP, KnX, X10 or Jini). They define an XML based Domotic Markup Language to represent devices in a uniform format and content on the DomoNet framework.

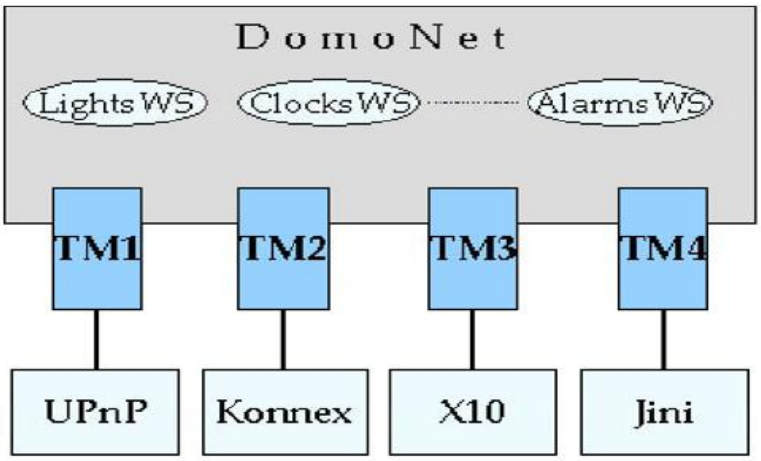


Figure 7.8: The DomoNet Architecture [Miori 06]

Figure 7.8 shows the DomotNet architecture. The *TechManagers*, one for each protocol represent each device as a virtual device on the DomoNet framework. The virtual device is exposed as a DeviceWS (Web Service) using the DomoML format and semantics. The LightWS, ClockWS shown in the figure are DomoNet standardized device profiles represented in the DomoML language. Thus, each device profile defines a set of functionalities a virtual device is supposed to support. For instance, a LightWS device supports two functionalities the Switch on/off functionality and the change light intensity.

The TechManager also reifies the virtual device as a real device. For instance a LightWS which represents an X10 light is also represented as a UPnP Light in the UPnP network.

Other similar approaches have been proposed in the literature such as the uMiddle [Nakazawa 06, Nakazawa] and the Web Services on Universal Networks [Yim 07] Frameworks.

7.4 Comparison & Discussion

In this section, we outline a comparison (see Table 7.1) between the previously overviewed approaches already classified in the three following categories:

Common Ontology used to capture all the semantics of a domain. The common ontology is built by an ontology expert. The interoperability between devices or services is possible only if the description is annotated by the concepts of the common ontology.

Abstract Model allows to manipulate services' interactions and behavior through high level concepts. Then using MDE techniques and transformation rules specified by designers, these high level concepts can be automatically transformed into an executable code which provides interoperability.

Unified Language/ Interface proposes a common description format and content to represent devices and their services through a unified description format and semantics. This category requires maintaining a mapping between various heterogeneous description and the unified proposed representation. The table between the local descriptions and the unified description is maintained by a domain expert which manually proposes the correspondences.

We present in Table 7.1 a comparison of the previously detailed interoperability approaches based on the following criteria. The various **protocols used** in each approach in order to interconnect the devices and applications, see Table 7.1. Then, for each approach we expose how the heterogeneity problem is tackled at the three layers detailed in chapter 6: the **protocol stacks mediation**, the **presentation** format and the **content** mediation. Additionally, for each layer, we outline the module used in each approach in order to resolve the heterogeneity problem. Furthermore, we depict the overall **realization** techniques used in each approach. We also take into consideration during the comparison the **description annotation**, whether (or not) the services' descriptions need to be manually annotated in order to resolve the heterogeneity problem at the presentation and content layers. Finally, we outline the **human intervention** in the adaptation and interoperability process.

Paolucci's approach along with the MySIM and PERSE middlewares use a common ontology to capture the domain semantics. This category shares the four following common points.

- A common ontology needs be specified by an ontology expert in order to capture the domain. Additionally, common semantics need to be found between relatively common concepts. In section 5.1.2, we outlined that the ontology development methodology is a difficult task, where multiple iterations are necessary until a correct version with coherent semantics can be found. Besides, building a common ontology for relatively complex devices can be more difficult and error prone. In the appendix B.2, Figure B.1 shows the *PrintTicket* element, an input parameter used to invoke the action *CreatePrintJob* in order to print on a standard DPWS printer.

- The common ontology is used to resolve the content heterogeneity of descriptions by representing concepts using a common syntax in a common taxonomy along with the relations between.

- The realization of the interoperability involves an ontology reasoning which classifies the concepts in a sort of a taxonomy along with the shared elements and properties. After the reasoning, comes the matching between the service categories, input and output parameters. Then, the adaptation process can be carried out through different techniques such as bundle generation.

- The service description must be annotated with elements used in a common ontology (annotation can also refers to other ontologies). Thus, a sort of a manual mapping is first applied by a human expert

on the service description by employing the same concepts from the common ontology. Additionally, the common ontology might need to be extended and updated if the concepts of the service description are not available in the current version of the common ontology. The update process can also be difficult and requires resolving conflicts where two concepts might have crossed semantics.

Paolucci's [Paolucci 02] approach deals with the service matching between web services having WSDL descriptions annotated with concepts from a common ontology. Thus, obviously there is no protocol or representation mediations. Paolucci do not presents an adaptation realization but focuses only on the service matching.

PERSE [Mokhtar 07] and MySIM [Ibrahim 08] both use a common ontology as a taxonomy holding the concepts in a hierarchical classification. MySIM focuses only on OSGi services, thus eliminating protocol use and mediation issues. It uses a translator module to represent OSGi service description in a common model. Then, the content heterogeneity is resolved through reasoning and matching of services' descriptions using the common ontology.

MySIM supports the service transformation and composition. The transformation usually adapts parameters while the composition relies on existing services to propose new capabilities. New bundles are created to realize the adaptation. MySIM first introspects the byteCode of the existing services' bundles, then the byteCode is replicated in the new bundle in order to add the translation behavior, the redirection mechanisms or to compose new services.

PERSE [Mokhtar 07] targets the UPnP and SLP protocols. It also translates the service descriptions into a common model, then applies reasoning and matching algorithms. PERSE provides in [Mokhtar 07] a detailed theoretical and well defined methods to achieve adaptation and service composition. However, the realization of such adaptation is not available in their approach.

Additionally, both MySIM and PERSE tackle the QoS during service matching and adaptation. In our context, we discard this feature, since the actual device and service description do not provide any QoS information.

The second category uses an abstract model which allows experts to manipulate high level concepts to provide interoperability. This category shares the two following common points:

- In this category, the three approaches propose high level models to conceptualize the main entities of a domain. This high level of modeling allows designers to manipulate concepts without facing technical requirements and constraints. However, this high level conceptualization requires from experts instantiating each device to be used. The instantiation includes providing information such as the device address, the device type, its supported services and actions using a high level language. Thus, each new device appearing in the network cannot be handled unless its instantiation is present.

- Two kinds of transformation rules must also be specified by experts. The transformation rules from one model to another which correspond in our context to the behavior of the proxy. The proxy's behavior is usually specified using a high level behavior language. For instance, a proxy resolving the heterogeneity between two or more devices holds behavioral information on how to react to events and messages by applying redirection techniques or applying modifications on the messages content. Thus, the first type of transformation rules specifies the translation from one model to another mainly detailing the behavior of adaptation entities such as proxies. Such rules are usually written manually or generated using graphical tools using specific languages like ATL [Jouault 08].

  The other type of transformation rules is the code transformation rules which also needs to be specified by an expert. Such rules enable the automatic generation of high level concepts into a lower level concepts such

Middleware	Protocols Used	Protocol Mediation	Presentation Mediation	Content Mediation	Realization	Description Annotation	Human Intervention
Paloucci	Web Services	χ	χ	Common Ontology	Reasoning, Matching	Required	Ontology Specification, Service Annotation
MySIM	χ	χ	Translator	Common Ontology	Reasoning, Matching, ByteCode Introspection, Bundle Generation	Required	Ontology Specification, Service Annotation
PERSE	UPnP, SLP	Service Invocation	Translator	Common Ontology	Reasoning, Matching, ?	Required	Ontology Specification, Service Annotation
DOG	KNX, myOpen, X10	Network Drivers, OSGi	Network Drivers	Common Ontology	Reasoning, Message Generation	Not Required	Ontology Specification, Device Instantiation, Transformation Rules
EnTiMid	UPnP, DPWS, myOpen, Knx	Network Adapters	Service Generator	?	Bundle Generation	Not Required	Device Instantiation, Transformation Rules
PervML	Web Services, EIB, UPnP	Base Drivers, OSGi	Designers	Designers	Model to Bundle Generation	Not Required	Six Modeling Steps, Transformation Rules
HomeSOA	UPnP, DPWS, Bonjour, IGRS, X10	Base Drivers, OSGi	Refined Drivers	$\approx$, Smart Device	Service Discovery, Service Registration	Not Required	Interface Mapping, Universal Semantics
UMB	UPnP, Jini, LonWorks	UMB Adapter	UDT-LDT Mapping	UDT-LDT Mapping	Service Discovery, Service Registration	Not Required	Mapping Table, Universal Semantics
DomoNet	UPnP, KNX, X10, Jini	TechManagers	TechManagers	DomoML	Service Discovery, Service Registration	Not Required	Mapping Table, Universal Semantics

Table 7.1: A Comparison between Interoperability Middlewares (✓:Supported, $\approx$: Partially Supported, χ: Not Supported, ?: Information Not Clearly Available)

as Java classes and executable code. The transformation rules to automatically generate high level models into executable code are written once and used in each code generation. However, the transformation rules of behavioral rules to adapt from one model or service to another must be specified for each device or service type.

DOG [Bonino 08b] tackles the heterogeneity of home automation protocols such as the KNX, myOpen and X10 protocols. DOG uses specific OSGi network drivers similar to the base drivers detailed in section 3.3 which handle the protocol and presentation mediations. DOG uses the DogOnt ontology as a taxonomy like in the common ontology category and also uses it as an abstract model. DogOnt models functionalities such as dimming or switch apart and then it is associated to a device in the ontology. The human intervention occurs at three levels the ontology specification, the device instantiation ad transformation rules specification. The realization is carried out by reasoning and specific message generation to provide interoperability between devices without specific bundle generation.

EnTiMid [Nain 08] represents devices like myOpen and Knx as plug and play devices (UPnP and DPWS) through models. They use base drivers like network adapters to resolve protocol heterogeneity. The service profiles generation handles the representation format. However, it is not clear if the content mediation is resolved, i.e. if a KNX light is exposed as a standard UPnP light with standard content and semantics. The EnTiMid realizes the adaptation by applying transformation rules from a model to model EnTiMid to UPnP and from high level to low level code using the bundle generation.

PervML [Serral 08] follows a different approach, where analysts and architectures specify through 6 steps, as shown in Figure 7.5, all the components to be used in an application. They specify using high level language, the devices, the services and actions used along with the behavior of each component involved in the adaptation. Then, using code generation techniques, the high level modeling is transformed into executable bundles. However, the PervML approach does not take into account mobility and dynamicity of devices. It supposes that the devices are always present in a room such as a meeting room.

The final category uses a common language to expose the representation format and content in a unified semantics. This category shares the three following common points:

- Universal representation format and content are re-proposed by an expert in order to achieve a unified format and content. The proposed universal semantics tries as much as possible to capture semantics of the various device and service heterogeneous content.
- A mapping is maintained by an expert between the universal and the local content representations. The mapping includes the device, service types and versions along with their supported actions names, input/output parameters names and types.
- The realization of the adaptation is based on the service discovery where specific modules discover the service or device appearance, then query the mapping table for its universal equivalence. Then a virtual device or a proxy exposes its description in the universal representation format and content.

HomeSOA [Bottaro 08a] handles heterogeneity between UPnP, DPWS, Bonjour and X10 protocols. Base drivers resolve stacks heterogeneity while another layer of refined drivers expose unified interface per device type and domain through smart devices. A smart device resolves the format representation by exposing the device and its capabilities using a WSDL description. However, services, actions and parameters names along with their types remain the same from the original description, thus the content heterogeneity is not resolved. Once a smart device is discovered by an application, a test of the action names and parameters is needed in order to invoke the desired action. A UPnP light and a DPWS light will advertise a dimmable-Light interface,

however, the actions names remain different, i.e. *Switch* and *SetTarget*. In [Bottaro 07b], the HomeSOA outlines a solution where a repository is used to store service matching. However, no technical and performance details are provided.

The UMB [Moon 05] approach also holds a mapping table between the universal and the local representation and content. On a local device discovery, an adapter discovers the device and registers a proxy using the universal representation and content.

DomoNet [Miori 06] adopts a similar approach to resolve the heterogeneity between various protocols. TechManagers handle the protocol and presentation heterogeneity while the DomoML tackles the content universal representation. They proposed a set of standard profiles to represent devices and their supported capabilities. The realization is carried out by the registration of a virtual device upon the real device discovery. A manual mapping table is maintained between the DomoML profiles and the profiles of other protocol devices. A virtual device is re-exposed by TechManagers in each sub-network protocol, i.e. a UPnP virtualized device appears as a DPWS, X10 device. In [Miori 10], they outlined a limitation of this approach due to data synchronization problem between the various virtualized devices appearing in multiple networks.

Each of the three categories offers advantages and drawbacks. **The first approach** is based on a powerful semantic representation which captures the main concepts of the domain and the relations between. However, the common ontology need to be manually built by an expert. In section 5.1.2, we outlined that the ontology construction is an iterative and hard task. Moreover, plug and play devices can have complex representations, for example a DPWS printer has 2000 lines of WSDL description [Microsoft 07], Figure B.1 in the appendix B.2, shows an input parameter of the print DPWS action. The other drawback resides in the annotation of the plug and play services with concepts from the common ontology. Two options related to the service annotation appear. This first is to impose to competitors such as manufacturers and standardization committees a common ontology where its concepts are used to annotate devices' and services' descriptions. Obviously, this option is too optimistic, there had never been a unified description in the proposed standard profiles (UPnP, DPWS and IGRS) for the same device type. The second option consists in applying a manual annotation on each device description, i.e. the expert manually annotates the description by applying a mapping between the semantics of the description and the concepts of the common ontology. However, annotating the description manually can be difficult and error prone. Besides, new devices holding new concepts not yet represented in the common ontology, require adding a new device concept to the common ontology and connecting it to the other existing entities. The update can produce an incoherent ontology since a new type can have cross semantics with more than one existing concept [Noy 04].

The second category allows a manipulation of high level concepts by abstracting technical and implementation details. Thus, it allows to tackle the interoperability easily. A model to model transformation rules allow to specify the adaptation behavior between two services or actions. The main strength of this approach is the transformation of high level concepts into lower level executable code. The main draw back resides in the modeling process which can be composed of up to six modeling steps as shown in Figure 7.5. The modeling process includes an adaptation behavior specified manually by an expert to allow a correct adaptation between devices and services.

The final category proposes a unified representation format and content. It consists of maintaining a manually established mapping table between local and the unified proposed representations. The main drawback of this approach resides in the manual established mapping between descriptions. In this category, the main advantage consists in using a unified representation format and usually (in some works) a unified content representation.

We believe that an adaptation approach can be inspired from the three categories and benefits from their advantages. For instance, the ontology allows to semantically represent information of a domain and captures the relations between the expressed entities. Furthermore, the models offer the ability to manipulate abstract concepts independently from the technical details. Additionally, a set of transformation rules can be applied on the models. Such rules allow to go from one model to another on the same level of abstraction, or even from a high level to a low level concepts containing more technical details.

However, as mentioned before, building an ontology is a hard and an iterative task which can be error prone. Therefore, reusing ontologies is recommended instead of building new ones from scratch [Noy 01, Wache 02]. Moreover, we also outlined the effort needed to propose the models and the transformation rules through a time consuming process and iterations, see Figure 7.5. However, another alternative, the ontology matching [Euzenat 07] seems to enable detecting transformation rules and mappings semi-automatically through heuristic based algorithms. Thus, we provide in the next chapter, the ontology alignment techniques and algorithms that can be used to achieve a semantic interoperability.

Chapter 8

Ontology Matching

"It is of course unrealistic to hope that there will be an agreement on one or even a small set of ontologies ... . We will still need to map between ontologies"

– Noy [Noy 04]

Contents

8.1 Matching Techniques **109**

8.2 Ontology Alignment Tools & Frameworks **112**

8.3 Conclusion **114**

Ontologies have proved their advantages in the data mediation and integration from various heterogeneous sources such as databases and web pages. The main strength in using an ontology resides in its ability to capture semantics of a domain through a representation of the domain's main concepts and the relations between. Moreover, the ontology reasoning allows to infer new information from what is already present. Thus, as long as the various information sources refer to the same semantic concepts in a common ontology, the integration problem can be resolved by reasoning and extracting information to achieve interoperability.

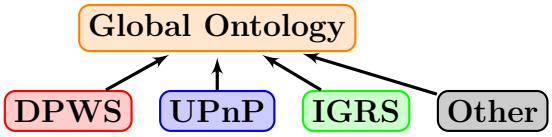


Figure 8.1: Integrated Approach

The works of [Ibrahim 08, Mokhtar 08, Paolucci 02] adopted the common ontology approach, also known as the integrated approach in the literature [Wache 02]. In the integrated approach, a global ontology is used, as in Figure 8.1, to represent other domains (UPnP, DPWS, other). Correspondences between the global ontology and such domains are established manually. The main challenge remains in the construction which is an iterative task [Wache 02] since common and unified semantics need to be found to represent heterogeneous resources. However, a central ontology will never be large and compatible enough to include all concepts of interest of every domain [Noy 04], so it will have to be updated, modified, extended and even matched with another ontology [Noy 04]. Each new extension will be different and can create conflicts between predefined concepts and semantics resulting with an inconsistent ontology [Wache 02]. Besides, an update in any domain requires an update of the global ontology.

Moreover, real world experiences demonstrate that a central common information model capturing all the semantics is not realistic [Euzenat 07, Noy 04]. The integration problem exists in different domains where each representation specified by different experts is specific to a certain application or context. Furthermore, as mentioned in chapter 5, the knowledge representation is imperfect and captures only a part of the reality. Thus, each representation will be oriented and built by domain experts to capture a part of the reality. Additionally, a common ontology is usually expressed in specific concepts used in a predefined jargon from a specific domain. As a result, different ontologies emerge and the need of re-usability and extension of ontologies lead to the ontology mapping which is proposed by the alternative known as the federated approach.

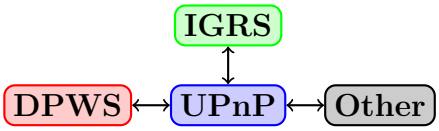


Figure 8.2: Federated Approach

In the federated approach [Noy 01], no common ontology is used, only correspondences between different ontologies are established using ontology alignment techniques as shown in Figure 8.2. The federated approach supporters like *Euzenat* [Euzenat 07], *Uschold* [Uschold 04] and *Noy* [Noy 04] agree on the fact that a representation will never be large enough to capture all the concepts and relations of a domain. Thus, a mapping between representations is needed at some point, which can be achieved through matching techniques. According to Euzenat [Euzenat 07], a *"Matching is the process of finding relationships or correspondences between entities of different ontologies"*, the result of the ontology matching is the ontology alignment representing the correspondences between various ontologies. Such correspondences allow then heterogeneous ontologies represented by different semantics and jargons to be interoperable. The detected correspondences between ontologies represent the transformation rules to go from one ontology to another which leads to information integration from various resources. The challenge in the federated approach consists in finding correspondences between the pivot and each domain. Such correspondences can be detected using the alignment techniques which are semi-automatic and based on the syntax, the semantics and the structure [Euzenat 07]. However, the alignment techniques are heuristic based, thus, a human intervention is still required to validate the detected correspondences. Since the mappings are independent, an update in a domain requires an update of the concerned mappings. If the pivot was updated then all the mappings have to be updated.

The federated approach is already employed in the semantic web and medicals domains such as in [Elbyed 09], to integrate data from different heterogeneous information sources like data base schemes and web pages. It is also used in some works for peer-to-peer interactions and web services composition where an interaction model is proposed to define the constraints that services have to satisfy such as in [Giunchiglia 06] and [Robertson 06]. However, to our knowledge the federated approach is not yet explored to resolve interoperability between Plug and Play devices.

Figure 8.3 shows an example of an alignment between two ontologies. Each value on the arrows refers to the confidence relation between the concepts. This confidence is referred to as the *Similarity* value (usually normalized) within an $\mathbb{R}^+[0,1]$ interval. The similarity value δ is calculated by various algorithms and matching techniques presented in the next section. A high value between two entities indicates a potential match according to the used matching technique. Some techniques calculate the normalized dissimilarity $\overline{\delta}$ which corresponds to $1 - \delta$.

We overview in this chapter, ontological matching techniques and tools proposed in the literature. In section 8.1, we outline different matching techniques. We provide in section 8.2, ontology alignment tools and

frameworks, then, we conclude in section 8.3.

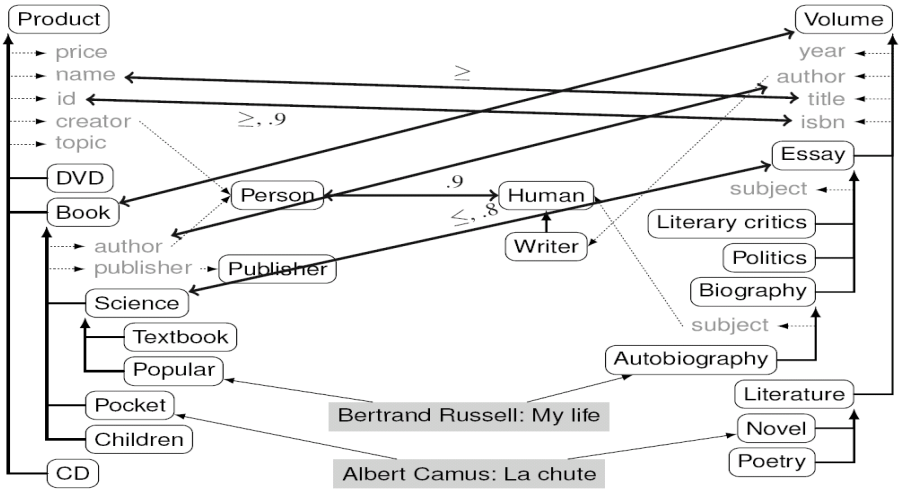


Figure 8.3: An Alignment Example Between Two Ontologies [Euzenat 07]

8.1 Matching Techniques

Various ontology matching techniques have been proposed in the literature [Euzenat 07] to detect correspondences between ontologies. Such matching techniques are based on various elements like the syntax, the language, the structure, and the semantics. For conciseness, we overview in this section only three categories.

Syntactical techniques are usually based on the comparison between strings. Before the string comparison, the string-based techniques apply normalization procedures to improve the matching results. Such normalization includes for example converting upper case letters to lower case (or lower to upper), or accents removal. Once the normalization is applied then one of the following string-based techniques can be used.

- The *Hamming distance* [Hamming 50] counts the number of positions in which two strings differ.
- The *SubString similarity* measures if a string can be derived from another string based on the prefix or suffix. For example, *LightDevice* is more specific than *Device*, therefore the *LightDevice* concept might be considered as a subconcept of *Device*. However, this technique can also give good results with non related strings such as the *article* string which is a substring of *particle*.
- The *Levenstein distance* [Levenshtein 65] calculates the minimum number of characters to transform one string into another using character insertion, deletions or substitutions.
- The String Metric for Ontology Alignment (SMOA) [Stoilos 05] technique is interesting in our context. SMOA is based on how computer researches and programmers choose descriptive names for the variables they use. Usually, they employ a mix of concatenating and trimming operations on strings such as *numPages* and *numberOfPages*. The general idea is to search for substrings then remove the common substring until none can be identified. For example, (*numberOfPages*, *numPages*) → (*numberOf*, *num*) → (*berOf*, ""). Then, the similarity value is based on the commonality between the two strings and the maximum common substring length. The commonality is calculated as follows:

$$Comm(S_1, S_2) = \frac{2 * \sum_i length(maxCommonSubString_i)}{length(s1) + length(s2)} \tag{8.1}$$

Thus, the SMOA similarity is the commonality value omitted from the difference value which represents the difference between the two strings. For example, the SMOA similarity value between *numberOfPages* and *numPages* reaches 0.91.

Other measures also exist such as the *Jaro* and the *Jaro-Winkler* based on characters' positions in the string. The *Path distance* can also be used to compare paths such as *Device/Service/Switch* and *Device/Service/Button*. This measure compares the strings' position and apply penalties when two strings are not at the same position in the path.

The string-based techniques can be used to rapidly detect relatively similar strings and mainly string extensions like a prefix or a suffix. However, if synonyms are used, for example, *article* and *paper* then it is obvious that such techniques will not detect any similarity. The linguistic techniques can be used to tackle synonyms and antonyms (down $\neq$ up).

Linguistic The language based techniques are based on the natural language processing (NLP) techniques. In [Euzenat 07], they are classified in two categories: intrinsic and extrinsic.

- The Intrinsic methods are based on linguistic processing techniques such as the tokenisation which allows to extract tokens by eliminating punctuations, digits and blank characters.
- The Extrinsic methods use external resources such as the WordNet [Fellbaum 98] dictionary. WordNet is considered as a large lexical database of English[1]. WordNet offers different relations between senses such as antonyms (up $\neq$ down) and synonyms where two words having different syntax refer to the same semantics. WordNet also proposes hypernym relation which is a (superconcept/subconcept) relation.

```
<Sense 1>: printer, pressman -- (someone whose occupation is printing)
 => skilled worker, trained worker, skilled workman -- (a worker who has
     acquired special skills)
  => worker -- (a person who works at a specific occupation ...)
   => person, individual, someone, somebody, mortal -- (a human being; "there
       was too much for one person to do")
          ...
<Sense 2>: printer --((computer science) an output device that prints the
    results of data processing)
 => peripheral, computer peripheral, peripheral device -- ((computer science)
     electronic equipment connected by cable to the CPU of a computer; "disk
     drives and printers are important peripherals")
  => electronic equipment -- (equipment that involves the controlled
      conduction of electrons (especially in a gas or vacuum or semiconductor)
   => equipment -- (an instrumentality needed for an undertaking or to perform
       a service)
    ...
<Sense 3>: printer, printing machine -- (a machine that prints)
 => machine -- (any mechanical or electrical device that transmits or modifies
      energy to perform or assist in the performance of human tasks)
  => device -- (an instrumentality invented for a particular purpose; ...
```

Listing 8.1: A Fragment of WordNet 2.1 Results for the word "printer"

Listing 8.1 reproduces the results provided by WordNet 2.1 for the word "printer". There are three senses, the first one refers to the printer profession, the results show that a printer and pressman are synonyms, and a printer is a subconcept of skilled worker, worker and person. WordNet also provides

[1]The EuroWordNet is another adaptation to other languages

a definition for each concept expressed in a natural language as shown in the Listing 8.1. The senses 2 and 3 refers to the word printer as a device and a machine.

Various techniques operates on the results of WordNet. Some techniques only exploit the synonyms relations provided by WordNet and consider that two concepts are equivalent if they are simply synonyms. Other techniques rely on the hypernym structure i.e. the subconcept/superconcept relations provided by WordNet to measure the similarity between concepts. For example, it can take into consideration the number of structural levels separating the two concepts. And some techniques take advantage of the definition provided in a natural language and base their similarity measures on the detection of the searched concepts in such definition. For example, in Listing 8.1, the sense 2 holds the word device in the definition of printer, thus a correspondence between device and printer can be established based on the definition provided by WordNet. A comparison of some WordNet based techniques is provided in [Budanitsky 06].

The linguistic techniques used with external tools such as WordNet allow to enhance the mapping between various entities based on the semantics instead of the syntax. However, the dictionaries like WordNet provide multiple senses for the same concept, thus proposing multiple potential matches. This can take place for example if the concept *printer* is present in an ontology $O1$ and the concepts *device* and *equipment* are both present in an ontology $O2$. Then, in general and according to WordNet both matchings are valid. In order to provide more accurate results, the structural methods can be used.

Structural based techniques relies on the properties between concepts. Two types of structural techniques exist in the literature. The internal structure based and the relational structure based.

- The internal structure based techniques rely on specific properties between the concepts. The data type property (string, int, float) can be used to determine which alignment to keep when two or more have relatively close similarity values. For instance, in Figure 8.4 since *id* and the *isbn* are both "uri", then a potential match is detected. It is obvious, that this technique need to be coupled with other techniques since the *price* and the *year* can also be considered as potential match.

  The cardinality of a concept can also be used to chose and decide about close alignments, for example some concepts can have at most 3 relations with another specific concept while others do not have any restrictions. Other properties can also be taken into account such as the transitivity and symmetry of a relation in an ontology.

  The internal structure methods can be used to determine or decide between two relatively close similarities. Used alone, such techniques are not efficient since two ontologies can have relatively similar internal structure but capturing different domains. Thus, the internal techniques must be coupled with the previous overviewed techniques.

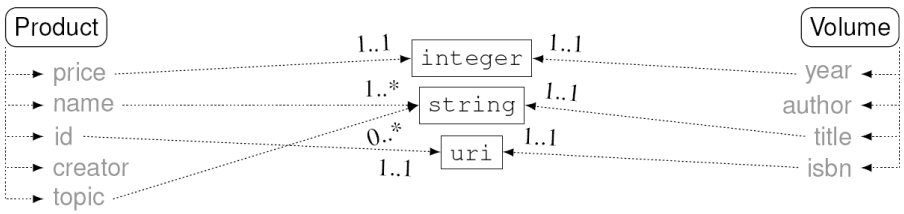


Figure 8.4: An Internal Structure Alignment Example Between Two Ontologies [Euzenat 07]

- The relational structure techniques take advantage from the taxonomic structure of the ontology when matching. In such techniques, the ontology matching is treated as a graph with arcs holding

relational properties between entities (SubClassOf, subPropertyOF, samePropertyAs), the matching compares concepts by taking into consideration their subconcepts, superconcepts and the number of arcs separating them. The similarity flooding algorithm [Melnik 02] is well known example of the relational structural algorithms.

The relational techniques are powerful since they take into account the relations and the structure between concepts, they are usually combined with the internal structure along with linguistic and the string based techniques.

Matching Strategies Each of the previous categories has its own advantages and drawbacks. The syntactical techniques allow to detect prefixes, suffixes and common substrings. Thus, they can be applied to obtain a quick evaluation on the similarity between two strings. However, their precision is not accurate since they are based only on the syntax. For example, the two strings *article* and *particle* which are not related will score a high similarity value using the syntactic matching techniques since they share the same suffix.

The syntactic matching techniques can be reinforced with the linguistic based techniques. The intrinsic methods can be used to normalize the strings to be matched. For instance, the intrinsic methods will remove punctuations or accents. Thus, the normalization can help to achieve better results since the punctuations and other non essential characters are removed.

Another advantage of the linguistic techniques relies on the use of external dictionaries which allows to detect synonyms or antonyms. Thus, the linguistic techniques are more accurate than the syntactic techniques since their metrics are based on a linguistic dictionary. Even though, the linguistic techniques provide more accurate results than the syntactical ones, however, the time to access and query a dictionary must be taken into account when dealing with large ontologies.

The structural matching techniques exploit the structure of the ontologies and take into account the number of childes and other criteria such as the parameter types and properties relating the entities together.

Since each category has its own advantages and drawbacks, these matching techniques can be combined in a global matching strategy to improve the ontology matching. For example, the techniques can be used together in a parallel composition. Each technique (k) can be assigned a weight α_k, which represents the accuracy to its calculated similarity value. Then, the similarity values of each techniques can be summed and normalized as follows.

Let $Sim_k(i,j)$ be the similarity result between the concept i from O_1 and the concept j from O_2 using the technique k. The average weighed similarity between the concepts is calculated as follows:

$$WeightedSum_Similarity_{i,j} = \sum_{k=1}^{n} (\alpha_k * Sim_k(i,j)) \qquad with \sum_{k=1}^{n} \alpha_k = 1. \tag{8.2}$$

A sequential composition can also be used to combine the matching techniques.

Several systems apply a combination of different techniques, such as in the ROMIE system [Elbyed 09] where the string-based techniques are used with the WordNet dictionary and a structural mapping technique which enforces the similarity values between the concepts.

Filters can also be applied on the alignment to keep only those with similarity values above or equal to a predefined threshold.

We overview in the next section some ontology alignment tools and frameworks.

8.2 Ontology Alignment Tools & Frameworks

Various ontology alignment tools and frameworks have been proposed in the literature, for conciseness we only detail two frameworks COMA++ and ROMIE, we also overview the alignment API allows the development of

alignment algorithms and matching techniques. Other alignment frameworks and tools are detailed, compared and evaluated in [Euzenat 07].

COMA++ The Combination of MAtching algorithms (COMA) tool is proposed in [Do 02]. COMA allows to match schemes such as relational tables or XML elements and structures. COMA starts by transforming the schemes into an internal format. The format represents the schemes as an acyclic graph, then matching techniques can be applied.

COMA supports three string-based techniques and a *soundex* matcher based on the phonetic between names, a *synonym* matcher which uses an external dictionary, and a *datatype* matcher. COMA also supports 5 hybrid matchers based on a combination of the previously mentioned simple matchers and other hybrid matchers. For example the hybrid *Name* matcher first applies pre-processing steps such as the tokenization, then it uses a set of simple matchers. The *NamePath* hybrid matcher applies a structural and name comparison. It first concatenate the names of a path into one single string then applies the *Name* matcher to calculate the similarity. The *Children* matcher takes into account the structure of the schemes. They apply a down-top matching where the similarity between two elements is computed from the similarity between their children. The COMA also allows users to validate and edit correspondences. A *Leaf* matcher is also supported to match instances.

The COMA++ [Aumueller 05] is an enhanced version of COMA, ontologies written in OWL are supported. Additionally, COMA++ provides a GUI to help users validate and edit ontologies. A taxonomy matcher can also be used where a specific domain taxonomy is used as an intermediary ontology instead of WordNet which holds a more general taxonomy.

Unfortunately, COMA++ is not an open source tool and only a prototype version with reduced features can be obtained for a limited time.

ROMIE [Elbyed 09] is based on the Ontology Mapping Interactive and Extensible environment (OMIE) [Elbyed 09]. The OMIE framework targets biomedical domain where large ontologies need to be accessed and mapped with other ontologies to retrieve additional information. The OMIE framework applies pre-processing normalization steps and uses a matching strategy combined of syntactical, linguistic, structural and semantic matchers. Each matcher is assigned with a confidence weight. OMIE uses two external resources when applying the linguistic matcher: WordNet and the Unified Medical Language System [2] a sort of a biomedical dictionary with specific vocabularies related to the biomedical domain. OMIE also provides structural matching algorithms with two modes top-down and down-top with different confidence degrees. The structural matching techniques takes into account when calculating the similarity value between two concepts the similarities of their subconcepts and superconcepts. OMIE [Elbyed 09] also supports a semantic matcher which takes into account the semantic relation between concepts such as the equivalent, symmetric and transitive relations. The ROMIE framework extends OMIE by integrating resource instances to enhance the matching process.

The ROMIE and the OMIE framework are not available publicly.

Alignment API [Euzenat 04, David 10] is an alignment tool which provides high level abstractions to manipulate ontologies. The Alignment API supports some basic string-based matchers such as SMOA, Levenshtein. It also provides an alignment format which encourages sharing and reusing alignments results. The alignment format contains references to the two compared entities (belonging to the two ontologies) and the calculated similarity value, see Listing B.3 in the appendix B.5. Additionally, the API provides

[2] http://www.nlm.nih.gov/research/umls/

alignment transformation from the alignment format into ontology OWL axioms. The transformation allows to add properties such as the *equivalentTo* property between two matched entities. The transformation result can be merged with the two ontologies used as an input of the alignment to produce a third ontology holding the matching between entities in a standard ontology language. Such merge allows other algorithms, techniques or reasoners to exploit the alignment result for example to retrieve information or apply some context or service adaptation.

The Alignment API is available publicly as an open source tool.

8.3 Conclusion

We provided in this chapter, the motivation behind the ontology matching to integrate information from various heterogeneous sources. Additionally, the ontology matching is considered more realistic than having an agreement on a common ontology.

We overviewed in section 8.1 three categories of the existing matching techniques available in the literature. The syntactical techniques are string-based and can be used to detect a matching between a string and its extension such as the prefix, suffix and some concatenation between two or multiple strings. The SMOA is an interesting syntactical technique in our context since it is based on how computer scientists choose entity and variable names. Such names are usually a concatenation of two or more strings. The linguistic techniques have two categories, the intrinsic methods can be used for a pre-processing step since they can eliminate non usable information such as the punctuations and the digits and outputs tokens to be used by other techniques. The second category of the linguistic techniques uses an external resource such as a dictionary, WordNet for example. The linguistic techniques allows to detect mappings between strings based on synonyms, antonyms and hypernyms. The structural based techniques represent the third category where the matching is based on the internal structure of the ontologies such as the datatype and other properties such as the cardinality. The structural techniques are based on the taxonomic nature of the ontologies (subconcept, superconcept) in order to detect alignments between entities. The alignment result can also depend on the alignment result of the sub-entities and the super-entities. Such techniques can be composed in order to produce a matching strategy.

We summarized in section 8.2, two alignment frameworks using various simple techniques and matching strategies. We also presented an alignment API which offers an alignment format and transformation techniques to exploit the ontology alignments.

In the next part, we detail our contribution and how these basic matching techniques and strategies assisted with rules can be used to provide interoperability between plug and play services.

Part III

Contribution

Chapter 9

Dynamic Service Adaptation for Devices' Interoperability

"Complexity can neither be created nor destroyed; it can only be displaced."

– Hurst's LAW

Contents

9.1 Motivation **117**
9.2 Overview **119**
9.3 An End To End Architecture **122**
9.4 Device Matching **130**
9.5 Concluding Remarks **152**

We detail in this chapter our approach to tackle the plug and play device and service interoperability. In the next two sections, we provide the motivation and the overview of our proposed approach. Our contribution is based on the automatic generation of ontologies capturing the device description which is presented in section 9.3.1. Then, we explain in section 9.4, our ontology matching strategy which consists of four steps to semi automatically find and validate correspondences between ontologies representing equivalent devices. In section 9.3.4, we present the global view of the architecture between the operator's platform and the end-user's digital home with regard to our proposed adaptation modules. And finally, we describe in section 9.3.3 the architecture of the automatic proxy generator and how we exploit the ontology alignment to generate specific proxies to provide a transparent plug and play interoperability.

9.1 Motivation

In this section, we overview the main motivation behind our approach with respect to the advantages and drawbacks of the various approaches proposed in the literature tackling the device and the service heterogeneity. Thus, we briefly revisit the three types of the proposed solutions, detailed in chapter 7. Furthermore, we depict their limitations in the light of chapter 8 which proposes a semi automatic alternative to find correspondences between descriptions in order to provide interoperability.

The common ontology offers a powerful representation language to capture the semantics of a domain. It allows to represent the main concepts and the relations between. The inference capability and the taxonomic

structure represent the main strength points in this category. Those two points allow to automatically extract information from what is already represented in order to provide semantic interoperability.

However, in chapter 5, we overviewed the ontology development methodology and showed that such task is tedious, error prone and requires multiple iterations until unified concepts can be found and agreed upon.

Furthermore, the proposed solution in this category requires a manual annotation of the service description exclusively with services from the common ontology. Thus, the description annotation is a manual mapping task, which relies on a first step where an expert must understand the semantics of the service and device description. Then, the second step consists in identifying the service, actions and parameters categories presented in the description and associate them with the concepts from the common ontology. The final step is the manual description annotation with the identified concepts from the common ontology.

Additionally, in chapter 8, we pointed out that a common ontology will never be large enough to capture a domain, therefore the common ontology must be extended to represent additional information included in some service or device description. Moreover, the new integrated concepts can hold crossed semantics with the already represented concepts transforming the ontology into an incoherent structure. Furthermore, at some point the integration effort becomes too tedious to integrate it manually since additional common semantics must be found. Though, the integration can include a reuse of an already developed ontology by other experts to add additional concepts.

Furthermore, information integration experts agree that at some point, the ontology mapping is needed. In fact, various ontologies will be proposed and developed by experts choosing concepts from their domain specific jargon. In chapter 8, we presented the ontology matching techniques which can be applied to semi-automatically find alignments between ontologies.

Thus, from this category, we retain the following: ontologies represent a powerful tool to represent information. However, developing ontologies manually must be avoided, we should privilege instead reusing already proposed ontologies and explore semi-automatic alignment techniques.

Abstract model allows to manipulate high level concepts independently from the technology used underneath. Thus, the interoperability can be tackled using high level concepts using a high level language which specifies the behavior of a proxy device concept in order to achieve an interoperability between services. This specified behavior represents the transformation and the adaptation rules which allows two services to interact.

The main strength of this category resides in the automatic generation techniques offered by the model driven engineering domain which allows to go from high level abstract concepts to lower level executable code.

However, this category requires an expert intervention to instantiate the devices, services and actions from the abstract model and add additional information in order to be used once the code is executed. Additionally, two types of rules must be specified by an expert. The high-to-low level transformation rules are specified once and used to automatically generate code. The other transformation rules are those which allow to go from one model to another, i.e. the translation and behavior rules which provide interoperability. The behavior and translation rules must be specified by an expert for each interaction between services and devices.

Thus, from this category, we retain its two major strengths: manipulation of high level concepts and the automatic generation of an executable code to provide interoperability. The manual specification of

the interoperability behavior rules can be probably reduced through semi-automatic ontology alignment techniques.

Unified interface/language exposes the representation in a unified interface or language to resolve the heterogeneity. A mapping table is maintained between the local descriptions and the unified one. Thus, applications will transparently interact with the adapted devices by using the same interface. The main strength of this category is its unified language to resolve the representation format and content. However, the mapping table is manually established by an expert.

Thus, from this category, we retain the unified language representation to resolve heterogeneity. The manual effort for the table mapping between the local and the unified might be reduced probably through semi-automatic ontology alignment techniques.

Based on the previous analysis detailing the advantages and drawbacks of each category, we select the major strengths in order to establish our contribution which is overviewed in the next section.

9.2 Overview

In our proposal, we took the best of the three previously detailed categories to tackle the plug and play device interoperability. The Ontologies allow to represent information of a domain through defined relations and concepts. The second category uses models and transformation rules to automatically go from high level to low level. The third category offers interoperability using unified semantic concepts. The approaches proposed in the related work requires an expert intervention and a manual effort to resolve heterogeneity and achieve interoperability. Thus, we would like to minimize the expert intervention in the adaptation process.

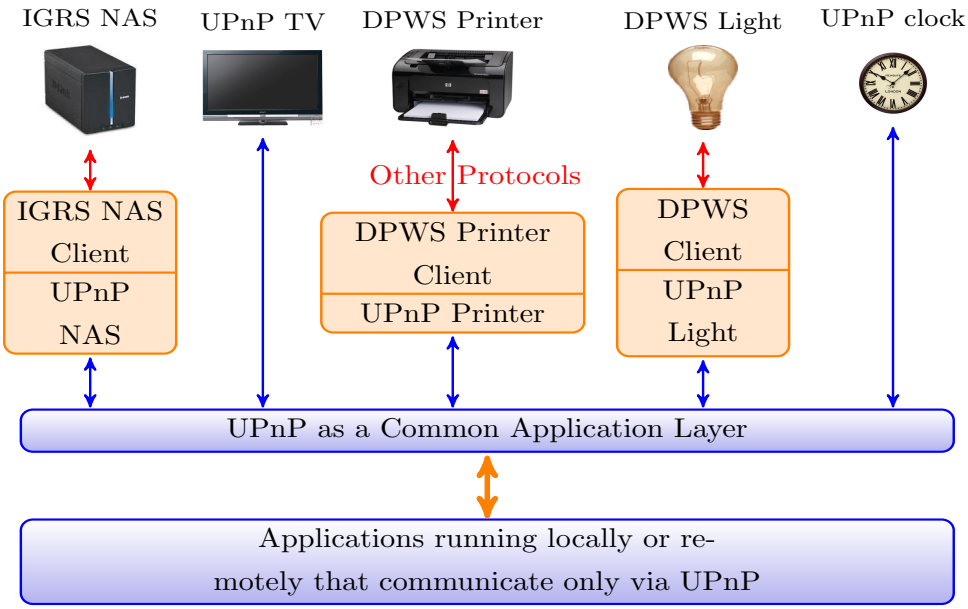


Figure 9.1: UPnP as a Common Application Layer

To accomplish interoperability between plug-n-play devices, we propose to use the UPnP representation content, format and stack as a common pivot, due to its wide acceptance among device manufacturers and vendors. Moreover, a large set of tools and applications targeting UPnP devices already exists [DLNA 03]. Additionally, the number of UPnP device types standardized by the UPnP Forum is the highest among the

other protocols. There is 26[1] UPnP standard device type while DPWS has only 3 standard types. IGRS relies on the UPnP standards while Bonjour has a limited standard devices like printers and the audio sharing devices *i*products. Additionally, as mentioned in section 2.4.3, the *BroadBand Forum* promotes the CWMP standard protocol [Broadband 10a] which can be coupled with a CWMP-UPnP bridge [Broadband 10b, Lupton 07] to offer a standard remote management solution adopted by many telecoms operators and device manufacturers [Broadband]. Such solution allows to remotely administrate from the operators' platforms the end-users devices located in the digital home. Bridging a remote administration protocol with a local plug and play protocol offers the telecoms operators, the device manufacturers and the end-users a lot of advantages [Lupton 07, Broadband 10a, UPnP 11] such as the remote diagnostic, troubleshooting and update operations. As for the other protocols, DPWS can be coupled with WS-Management protocol promoted by the DMTF [DTMF] , however it is so far mainly adopted by Microsoft and Schneider Electric. Bonjour and IGRS use proprietary administration protocols.

The UPnP is chosen as a common pivot in this thesis, however, our proposed solution is generic and another pivot can also be chosen instead.

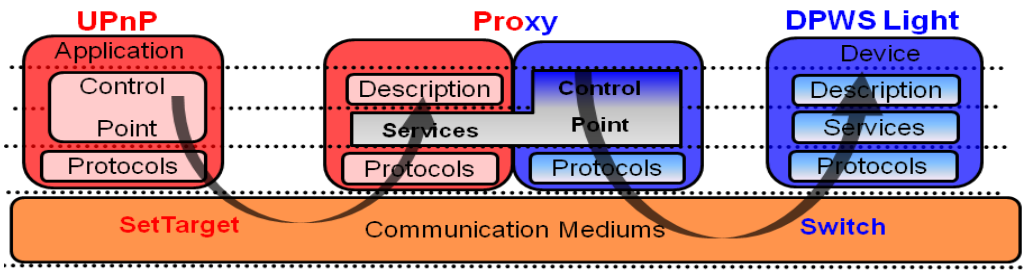


Figure 9.2: Architectural View of the UPnP-DPWS Proxy

Our approach consists in generating proxy modules, published as UPnP standard devices and able to control non-UPnP devices. Equivalent device types have almost the same basic services and functions, a printer is always expected to print and a light is expected to be turned on or off independently from its underlying supported protocol. In fact, some manufacturers even propose the same device type in two different plug and play protocols versions. Thus, with the required adaptation, provided by a specific proxy on the three heterogeneity levels (protocols stacks, format and content), applications can interact transparently with equivalent devices. A generated proxy, as shown in Figures 9.1, allows applications to interact with non-UPnP devices as standard UPnP devices. For instance, a UPnP-DPWS Proxy Light, as shown in Figure 9.2, is exposed as a UPnP Standard Light and controls a DPWS Light through its supported DPWS client. The proxy light exposes an identical description as a standard UPnP Light. In Figure 9.2, the proxy exposes a standard UPnP action `SetTarget` and searches for a DPWS Light with a `Switch` action. When the UPnP `SetTarget (boolean true/false)` action is invoked on the proxy, it will translate the call and invoke the equivalent action `Switch (String ON/OFF)` on the DPWS Light. Using UPnP as a common model allows developers to focus only on implementing applications that use the UPnP interaction model.

Figure 9.3 outlines the architecture of our approach. The first (MDE) M0 layer represents the heterogeneous plug-n-play devices with their descriptions expressed in different formats. UPnP uses an XML template format while DPWS and IGRS use the standard Web Service Description Language (WSDL). Each description uses different semantics: UPnP uses devices, services, actions and state variables while the WSDL uses hosting and hosted services, operations and messages.

[1]http://upnp.org/sdcps-and-certification/standards/sdcps/

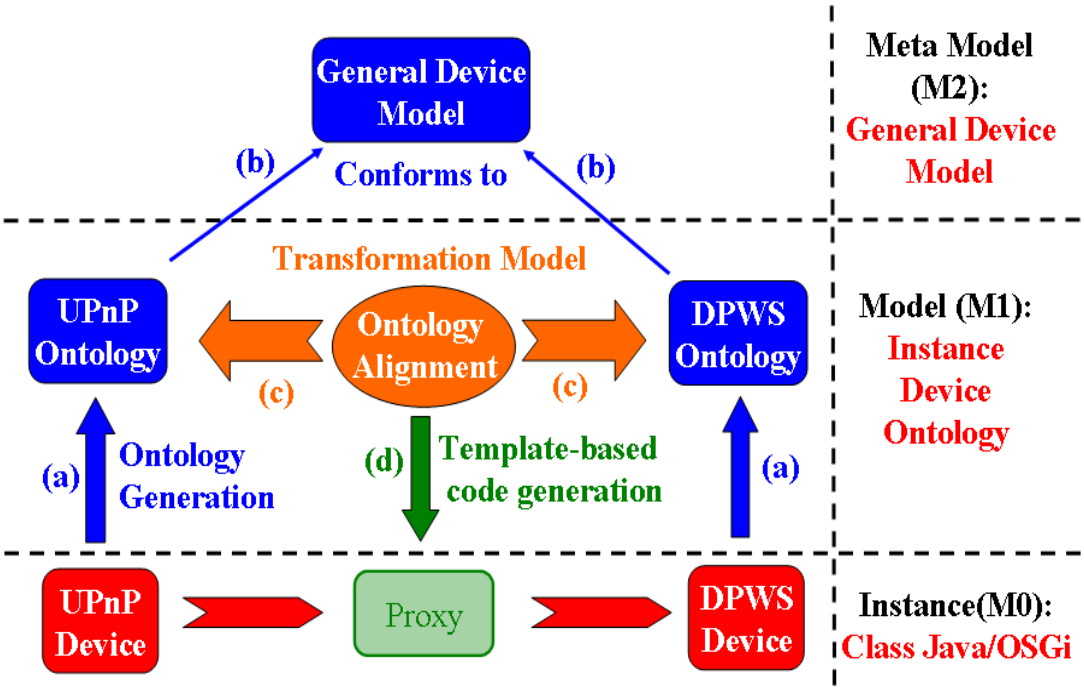


Figure 9.3: Overview Of The Approach

To resolve the representation format heterogeneity, we propose to automatically generate independent ontologies from each device and service description, instead of manually building common ontologies. The generated ontologies will resolve the representation format heterogeneity since the concepts from the device and service description will be expressed using unified representation concepts and relations. The ontologies represent the description content independently from the technology used and the local representation format. Thus, the automatic generation of ontologies lifts the device description from the M0 layer to the M1 layer as shown in Figure 9.3, the *(a) arrows*. Each ontology represents a device using unified concepts in conformance to the general device model in the M2 layer, the *(b) arrows*.

The ontologies resolve the description heterogeneity format in the M1 layer, therefore, in order to resolve the content heterogeneity, we apply ontology alignment techniques in order to find correspondences between two equivalent devices and their supported capabilities, as shown in Figure 9.3, the *(c) arrows*. The alignment techniques, as shown in chapter 8, are heuristics-based techniques, thus an expert intervention is required to validate the semi-automatically found alignments. Rules are also applied on the alignment to detect patterns in order to assist the expert during the validation. Additionally, the rules are used to automatically annotate the ontology with new properties expressing the detected patterns between the actions such as the composition relations.

Finally, once the ontology matching is validated between equivalent devices, the automatic code generation techniques allow to go from a high level independent technology representation in the M1 layer to an executable proxy code in the M0 layer, as shown in Figure 9.3, the *(d) arrows*. The proxy exposed as a standard UPnP device transfers the received invocations to non-UPnP devices.

The remainder of this chapter exposes our contributions which are two folds. An **end to end architecture** which provides the interoperability between plug and play devices and applications. The **device matching** which enables to semi automatically detect correspondences between equivalent devices.

Therefore, first, we present in section 9.3 our **end to end architecture** and detail each of its modules

in four subsections. We outline in subsection 9.3.1 our automatic ontology generation. Then, we overview in subsection 9.3.2 the device matching which allows to detect correspondences between two equivalent devices. In subsection 9.3.3, we describe the proxy generation from the previously detected correspondences. Then, subsection 9.3.4 shows the global architecture and how the proposed modules can fit in the digital home.

Then, we detail the **device matching** which relies on the ontology alignment and the different steps to achieve a valid matching between two equivalent devices. And finally, we end this chapter with some concluding remarks.

9.3 An End To End Architecture

We overview in this section our end to end architecture along with its three independent modules used in order provide a transparent plug and play interoperability between applications and devices. Figure 9.4 shows the modular architecture of our approach. The first module, *"Ontology Generation"* see Figure 9.4, handles the automatic ontology generation from the device and services descriptions. Then, the generated ontologies are used by the *"Device Matching"* module. This module takes two generated ontologies describing two equivalent devices type and applies heuristics based algorithms in order to find correspondences between the equivalent devices. Since the device matching module is heuristics based, an expert intervention is needed in order to validate the detected correspondences. Finally, the validated correspondences are used by the *"Proxy Generation"* module in order to automatically generate an adaptation code.

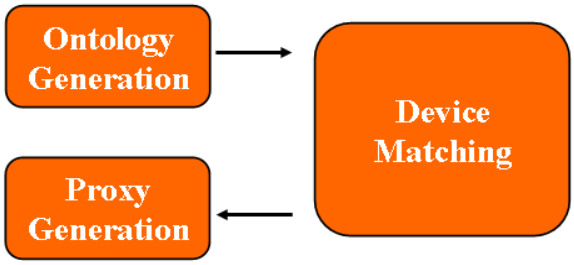


Figure 9.4: The Modular Architecture

We detail next each of the three modules: the **OWL Writers** module handles the automatic generation of ontologies while the **Device Matching** module applies the ontology alignment techniques and validation. And finally, the **DOXEN** module allows the automatic code adaptation which provides a transparent plug and play interoperability.

9.3.1 OWL Writers

Plug and play devices announce their device description and capabilities on the network, see chapter 4. In section 3.3, we outlined how base drivers can represent real devices as local OSGi services on the platform. Such OSGi representation allows local services on the platform to interact with the real devices on the network.

Thus, we designed software entities, the *OWL Writers* [El Kaed 10] as shown in Figure 9.5, one per protocol to generate ontologies based on the devices' descriptions. For example, a *UPnP OWL Writer*, subscribes to the arrival of UPnP devices. Once a UPnP device is detected by the UPnP base driver on the network, it is represented as a local OSGi service, as shown in Figure 9.5. Then, the *UPnP OWL Writer* is notified by the OSGi framework about its arrival, then the ontology generation is started. The *UPnP OWL Writer* parses the UPnP device description and generates an ontology conformed to a predefined meta model to represent the

device.

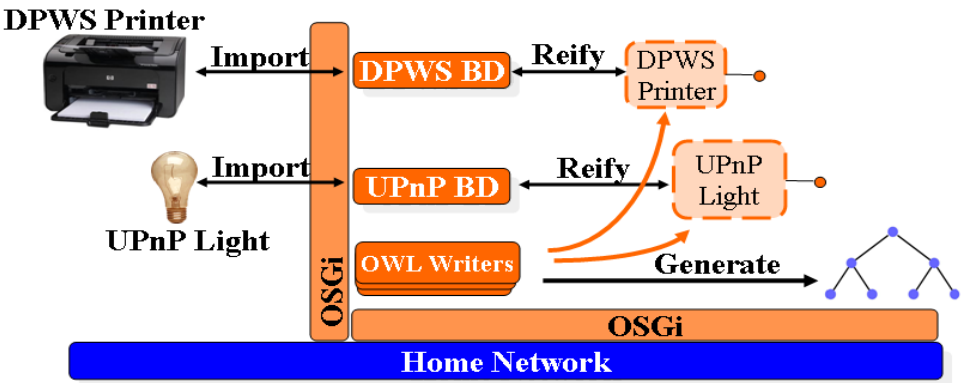


Figure 9.5: Ontology Generation by the OWL Writers

We define a general device model [El Kaed 10] as shown in Figure 9.6, inspired from the UPnP Meta Model [UPnP]. We use the concepts as follows: every device has one or multiple services, every service has one or multiple actions and each action has one or multiple input/output variables. A variable can also be directly related to one or several services. Additionally, each state variable has a type, a range and can have multiple default values.

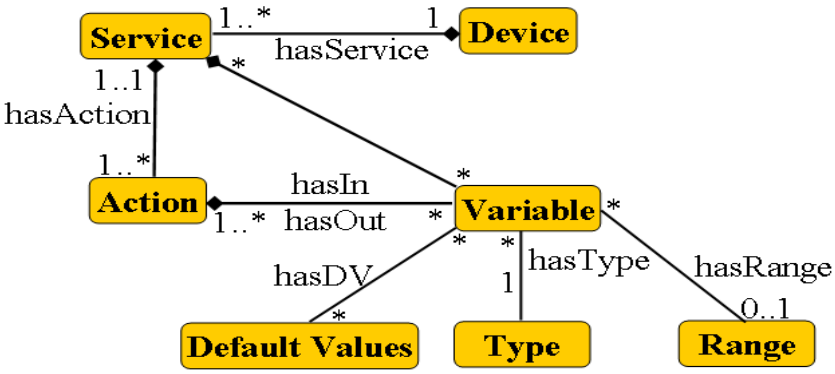


Figure 9.6: M2 Layer, General Device Model

The *OWL Writers* generate ontologies conformed to the defined general device model by construction. The ontology is expressed using the Ontology Web Language (OWL) [OWL]. Each ontology describes a device, its hosted services and actions along with the variables and their types. Figure 9.7 shows an example of an *intel* UPnP Light [UPnP 03] Device description and its ontology representation. Another example of a DPWS Light [SOA4D a] Device provided by the SOA4D project [SOA4D b] and its ontology are shown in Figure B.2, in the appendix B.3. Both ontologies use the same properties to describe and link the concepts. The ontology representation provides information about the concepts and how they are related. For example, the `BinaryLight` concept is related to the `SwitchPower` concept through the `hasService` property.

Figure 9.7 shows only the property view between concepts and how they are related. However, during the ontology generation, the taxonomic structure is also represented based on the device description which holds the necessary information to retrieve the concepts type. In Figure 9.8, the taxonomic structure view of the ontologies is shown. The left ontology corresponds to the UPnP light ontology of Figure 9.7 while the right part corresponds to the DPWS light ontology of Figure B.2 in the appendix B.3. Each concept used in the ontology

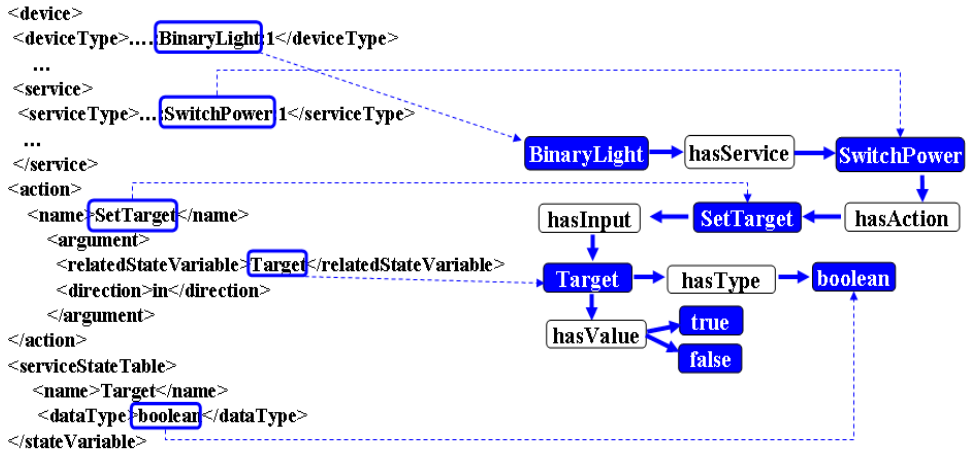


Figure 9.7: (Simplified) UPnP Ontology Generation from an XML description, (properties view)

in Figure 9.7 is a subconcept of a predefined concept. For example, the `BinaryLight` concept is a subtype of the more general `UPnP Device` concept. Thus, the taxonomy will provide additional information about the concept's type whether it is a device, service, action or variable. Figures 9.7, 9.8 show a simplified view of the ontology generation which is not a simple rewrite of the device and service descriptions. The ontology generation requires knowledge extraction from a non semantic representation usually expressed using XML and WSDL formats. More details on the ontology generation and the technical difficulties encountered are depicted in section 10.1 in the Implementation chapter.

Thus, each *OWL Writer* automatically generates an ontology which holds two types of information, the taxonomic structure of concepts and the relations between them expressed in unified semantics. In each generated ontology, the general concepts `Device`, `Service`, `Action`, `Parameter` and the relations between the concepts found in the device description are represented in unified properties such as the `hasService`, `hasAction`, `hasInput`, `hasOutput`, etc. The ontology represents the information in a high level (M1 layer) using high level concepts independently from the devices format description. An OWL writer also generates ontologies from a device file description statically, i.e. without the need to detect the device on the network and retrieve its description.

Figure 9.8: Part of the Taxonomic Structure of a UPnP (left) and a DPWS (right) Light

The *OWL Writers* can be placed on the clients' side, in digital homes, for example on a Set-Top-Box. Upon the detection of a new device, an *OWL Writer* checks if an ontology representing the detected device exists locally or on the operator's site. If it is not the case, then the *OWL Writer* generates an ontology and sends it to the operator. An *OWL Writer* can also be placed at the operator's site to generate ontologies statically

based on the devices' descriptions files.

We overview next the device matching module.

9.3.2 Overview of the Device Matching

The device matching module takes as input two generated ontologies which represent two equivalent devices descriptions, for example, a UPnP and a DPWS lights. Then, the device matching applies the ontology alignment techniques presented in chapter 8 along with other proposed methods detailed in section 9.4. Such techniques are heuristics based and allow to automatically detect correspondences between the two ontologies based on several criteria such as the ontology structure, the semantic names and number of elements. Furthermore, since the alignment is heuristic based, a human intervention is needed to validate the detected correspondences. Figure 9.9 shows a simplified part of the device matching result applied on two lights (UPnP and DPWS) ontologies. Equivalent entities from both ontologies are now related through the equivalence property. Figure 9.9 shows a simple mapping relations where an entity is equivalent to only one another entity. However, other union mappings exist which are detailed in the device matching module in section 9.4. The *equivalentTo* property connecting the entities in Figure 9.9 represent the semi-automatically detected translation rules to go from one ontology to another.

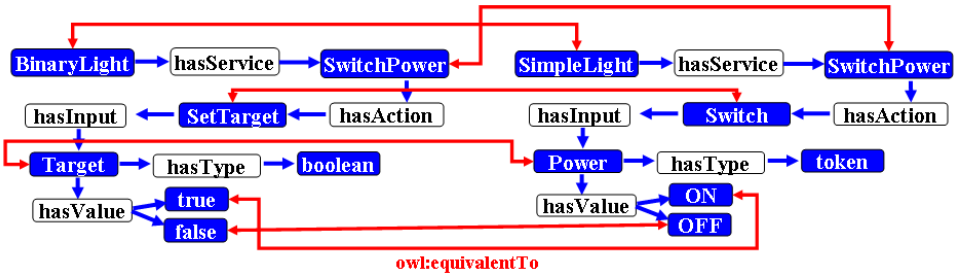


Figure 9.9: Part of a Lights Ontology

Now that the alignment is validated between the ontologies, the adaptation can start in order to provide an interoperability between the plug and play devices and services. The next section gives a detailed explanation about DOXEN and the proxy generation.

9.3.3 DOXEN

In this section, we detail our proposed DOXEN [El Kaed 11b] module, a **D**ynamic **O**ntology-based pro**X**y g**EN**erator which can be deployed on a *SetTopBox* for example. DOXEN automatically generates, a proxy for each valid equivalent non-UPnP device based on an ontology alignment. DOXEN takes as input an ontology alignment containing the equivalent devices' descriptions and the mappings between equivalent entities (devices, services, actions and variables). Based on the information in the ontology, DOXEN can generate using predefined templates, a proxy exposing a UPnP standard profile and able to interact with real devices having the same description as in the ontology alignment. The generated proxy is exposed as a UPnP standard device, it translates the action invocations to the real non-UPnP device.

The necessary steps carried out by DOXEN to generate and start a proxy are overviewed next. Then, section 9.3.3.2, points out the template filling principles to go from a high level representation contained in an ontology into a low level execution code.

9.3.3.1 Proxy Generation Overview

We describe in this section, the steps performed by DOXEN to automatically generate a proxy, as shown in Figure 9.10.

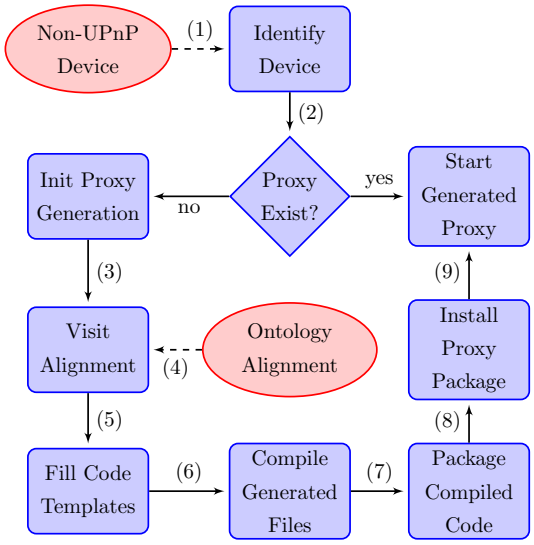


Figure 9.10: DOXEN Diagram Generation

DOXEN can be installed in the digital home. Once started on the set-top-box for example, it parses a configuration file containing information about the equivalent devices, the alignment file and the repository to download additional alignments. DOXEN listens to non-UPnP device arrivals on the network. When a non-UPnP device appears (step (1)), DOXEN is notified and receives its description and capabilities. Thus, DOXEN checks (2) the device type and version number along with its supported services. Then, DOXEN queries its local configuration and the operator's remote repository to check if the detected model is supported and has an equivalent UPnP device. If the model is supported, then based on the device type and version and its supported services, DOXEN loads the alignment ontology file (4) and walks through the equivalent entities and the detected patterns annotating the ontology. While exploring the ontology, DOXEN extracts from the ontology the information and instantiates the necessary objects.

Once, the ontology is explored, DOXEN starts the code generation by filling (5) the pre-written code templates based on the extracted information from the ontology alignment. Once the code files are filled and generated, DOXEN compiles the files (6) and builds a package (7) **at runtime**. Then it installs (8) and starts (9) the new generated package which corresponds to a UPnP proxy for the non-UPnP device. The started proxy requests the general device information (manufacturer, model, friendly name, ID, etc) from the non-UPnP device and publishes the same retrieved information during its UPnP format annunciation on the network. The end-user or the application identifies the proxy using the same type and the friendly name of the non-UPnP device. As soon as the non-UPnP device leaves the network, the proxy itself announces its departure from the network. When the non-UPnP device re-appears, DOXEN re-starts the proxy again which re-binds to the real device and re-announces its UPnP description.

The next step overviews the template filling steps (4) and (5) of the proxy generation.

9.3.3.2 Template Filling

DOXEN takes an alignment as an input, as shown in Figure 9.11, and uses predefined templates along with a compiler to generate executable code which corresponds to the proxy containing the behavior adaptation.

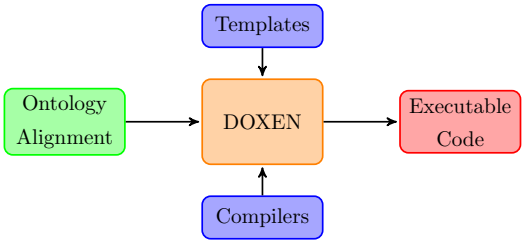


Figure 9.11: DOXEN

The templates consist of several files containing the implementation structure of a proxy. They hold the general behavior of a proxy when it comes to registering its services and announcing its description. Moreover, once started, each proxy searches for a specific non-UPnP device type and binds with it in order to translate the received UPnP actions' invocations to this real device. Thus, the search and the binding mechanisms performed by each proxy are supported in the templates. Additionally, extracting the parameters values from the received invocations along with the translation between values and parameters constitute common adaptation functions for all the proxies. Thus, such functions are also already supported in the templates. Furthermore, the actions adaptation and composition represent a general shared attitude between the proxies. Therefore, their implementation is embedded in the templates. The high level API code also exists in a file template and is injected in the code files when an expert adds a behavior adaptation to the ontology.

Thus, since the templates contain the general behavior of a proxy, they only need to be filled with specific information, such as the name of the device to search for, the equivalent services, actions and parameters. The template also needs to be filled with the composition relations between actions (simple, union, sequential union) and an adaptation code if necessary.

Figure 9.9 shows a part of an alignment between two lights and their equivalent relations. Moreover, the ontology contains unified relations which allow to go through all the concepts such as the *hasService* and the *hasAction* properties. The relations between semantically compatible entities are either the equivalence or the patterns properties detailed in the previous section. Based on such information, DOXEN walks through the ontology starting from the UPnP device concept and extracts the valuable information such as the entity names, the equivalent mappings and valid composition, then fills the templates. Each filled template will generate a file containing code ready to be compiled, packaged and executed.

The template filling allows to go from a high level representation model (M1 Layer) specified with an abstract representation language into a lower level (M0 Layer) language which is the code ready to be executed. The realization of such transformation relies on the content of the template, the ontology visiting and the template filling. The high-to-low level transformation is accomplished once and used for every proxy generation. In fact, the high-to-low transformation constitutes the main strength of the Model Driven Engineering domain promoting the *"write once and use anywhere"* slogan.

The next section overviews the global architecture between the operator's platform and the end-user's digital home with regard to our proposed adaptation modules.

9.3.4 Global Architecture

The previously detailed modules are used in the global architecture as shown in Figure 9.12. The OWL Writers detailed in section 9.3.1, automatically generate ontologies from devices' descriptions. Moreover, OWL Writers operate in two modes, either they are deployed in a network and when a new device appears, the OWL Writer discovers the Plug and Play device and then generates an ontology reflecting its description. The other mode is more static, using a shell command, an expert gives the device file description as an entry to the OWL Writer.

Thus, an OWL Writer can be deployed at the end-user's digital home. The OWL Writer will detect new devices and generate their correspondent ontologies when a non identified device is discovered. Then, the generated ontologies are uploaded to the operator's site. The OWL Writer can also be placed at the operator's site and will generate ontologies on the expert demand.

The Device matching explained in section 9.4 is performed on the operator's site through the ATOPAI framework. The expert retrieves from a database two OWL Writer generated ontologies representing two equivalent plug and play devices. The expert triggers the automatic alignment by selecting a matching technique. Then, the expert validates the alignment result though the ATOPAI's GUI. The valid alignment can then be deployed automatically or on demand on the home network in order to be used for the proxy generation.

DOXEN can be deployed on a set top box for example, it generates a proxy from each valid alignment which contains two devices profiles along with the mapping between. The generated proxy is then installed and started in order to resolve the heterogeneity between the two profiles. The proxy announces itself as a UPnP device and implement a client to interact with the non-UPnP equivalent Device. The proxy generation is triggered when an application is searching for a UPnP device, a UPnP printer for example and there is an available DPWS printer instead. DOXEN can also trigger the proxy generation when the DPWS printer appears on the network. DOXEN can also be installed at the operator's site instead of deploying it on the client's site. Then, the proxy generation will take place on the operator's site, the generated proxies will be remotely installed in home networks when a new device appears or on applications demands.

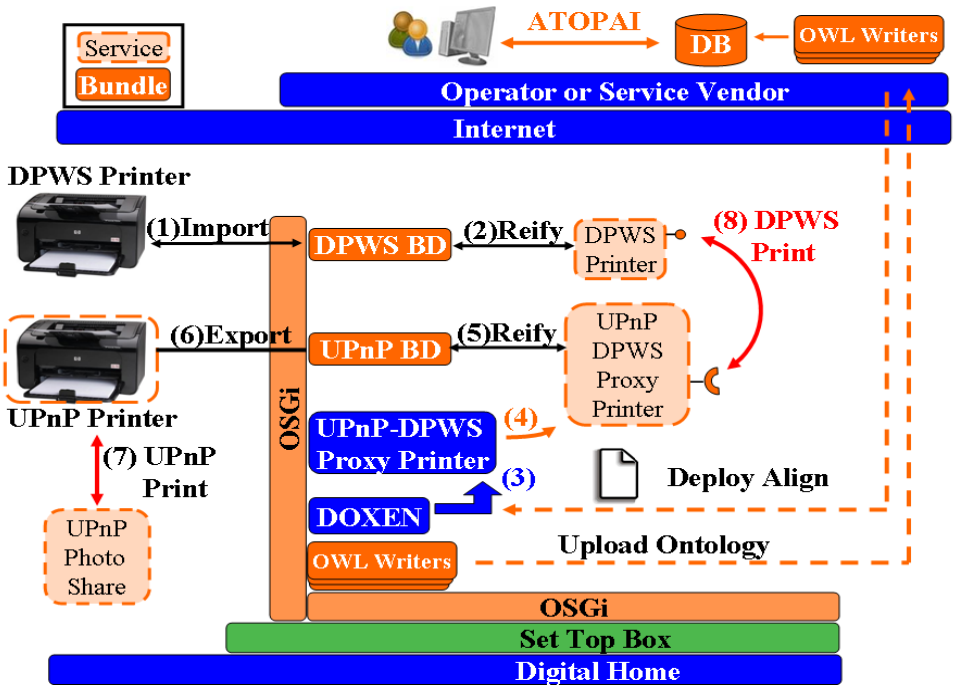


Figure 9.12: Global Architecture

Figure 9.12 shows the global architecture relating the operator's site and an end-user's digital home. DOXEN is installed on an OSGi framework on top of the Set-Top-Box. When a DPWS device appears on the network, the DPWS Base Driver discovers, imports (1) and reifies (2) the device as an OSGi DPWS Printer Device. DOXEN detects the OSGi DPWS Printer Device, checks the list of the equivalent DPWS devices. If the DPWS printer alignment is not present locally, then DOXEN sends a request to the operator's site. If the DPWS printer has an equivalent UPnP device and a valid alignment, then the operator deploys the alignment file on the Set-Top-Box. Once received, based on the alignment, DOXEN generates (3) a UPnP Proxy Printer (OSGi) bundle which publishes (4) the OSGi UPnP Proxy Printer Device.

The UPnP base driver is notified about the new UPnP Proxy Printer Device, therefore the UPnP base driver discovers it locally, and reifies (5) the new UPnP Device and exports it (6) as a UPnP device on the network. Any invocation (7) on the UPnP exposed device is handled by the UPnP Proxy Printer OSGi Service and forwarded (8) to the OSGi DPWS Printer Device. The invocation on the OSGi DPWS Device is reified by the DPWS base driver to the real DPWS Printer which executes the command.

When the DPWS printer leaves the network, the proxy receives the departure announcement and then sends its equivalent UPnP departure announcement on the network and then goes into an "OSGi" installed bundle state, see section 3.3.

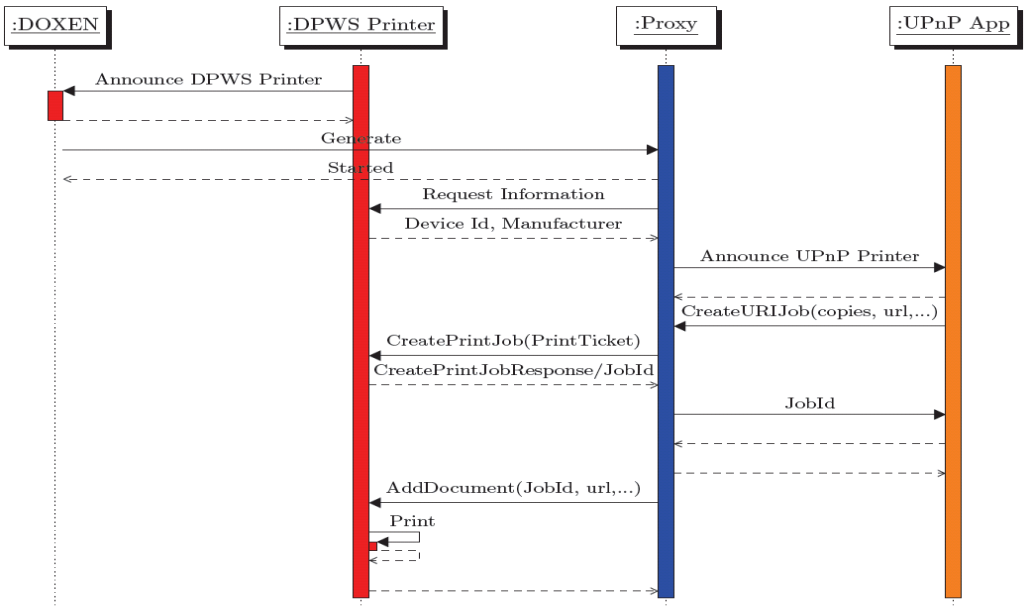


Figure 9.13: Sequential Diagram showing DOXEN and the Proxy Interaction

Figure 9.13 exposes the sequential diagram of the adaptation operation. First, the DPWS printer announces its description on the network. DOXEN already subscribed to the DPWS device appearance events, thus it will be notified when the DPWS Printer appears. DOXEN walks through the printer description, version and supported services. Then, DOXEN loads the specific alignment ontology, generates and starts the specific UPnP-DPWS proxy printer.

When started, the proxy searches for the DPWS printer and requests its general information such as the device ID, manufacturer, model, version etc and then uses the same information to announce itself as a UPnP Printer. For instance, if the DPWS Printer's model is *HP LaserJet 4515x* then the proxy will announce the same model and name. The UPnP Printing application receives the proxy annunciation and invoke the command *CreateURIJob* on the UPnP printer. The proxy receives the invocation, extracts the values set by the UPnP application such as the number of copies, and the document url and then applies the composition pattern

specified for the adaptation. The *CreateURIJob* as shown in Table 9.4 is equivalent to the sequential union of *CreatePrintJob* and the *AddDocument*[2]. Therefore, the proxy first adapts the received values and builds the input parameter of the action *CreatePrintJob* which takes as an input a *PrintTicket* complex structure shown in Figure B.1 in the appendix B.2. The proxy then invokes the *CreatePrintJob* action on the DPWS printer and waits for the return value. The value is extracted and returned to the UPnP application. The proxy builds the input parameter of the *AddDocument* action and invokes the DPWS printer which finally prints the document. The generated proxy contains the adaptation behavior to provide interoperability between two equivalent devices profiles. Thus, the proxy can be exposed as a UPnP printer and will interoperate transparently with any UPnP application searching to invoke a printer with standard UPnP actions.

The following section details the device matching which constitutes the second fold of our contribution.

9.4 Device Matching

The device matching as shown in Figure 9.14 is dependent on the ontology generation step. It takes as an input two automatically generated ontologies by the OWL Writers. The device matching allows through four major steps to semi-automatically detect translation rules between two devices' descriptions. We rely on the semi-automatic ontology alignment to detect correspondences, instead of specifying the adaptation rules manually. The detected adaptation resolve the content heterogeneity and allows to go from one description device to another.

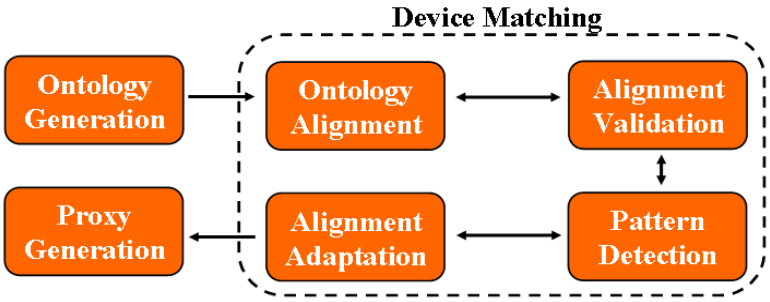


Figure 9.14: The overall major steps of the process

The device matching detects compatibility between two equivalent descriptions like two printer devices. Such compatibility is in fact a valid matching between the device services, actions and parameters. In other words, if the device matching is valid, then using a particular proxy, the device can be substituted with its equivalent announcing a different description and using a different protocol. The proxy will handle the redirection and the invocation translation with the required parameters.

Figure 9.15 shows the whole process of the device matching and details the four steps of Figure 9.14. The process takes two automatically generated ontologies (Ont1, Ont2) by the *OWL Writers*. The two ontologies correspond to two equivalent devices' descriptions such as a UPnP and a DPWS printers. First, the ontology "Alignment(1)" techniques and algorithms are applied in order to automatically detect correspondences between equivalent entities from the two ontologies, see Figure 9.15. The ontology alignment techniques are detailed in section 9.4.1. This step produces an alignment file (.rdf) containing several tuples $< leftEntity, rightEntity, SimilarityValue >$ referring to left and right entities from both ontologies along

[2]The AddDocument takes the document URL location as an input. The SendDocument can also be used, the document is then sent to the printer as an attachment. In our experimentation, we used the more complex adaptation using SendDocument

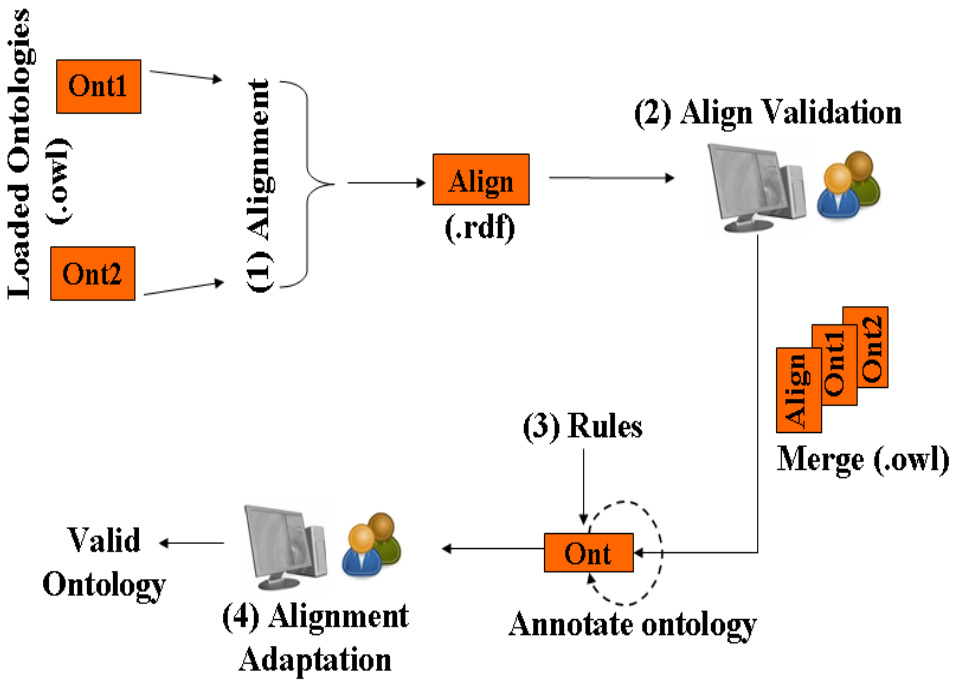


Figure 9.15: The Device Matching Process Overview

with their calculated similarity values. As detailed in chapter 8, the similarity value corresponds to the matching confidence between the two entities.

However, since the matching is heuristics based, an expert intervention is needed to validate and edit the semi-automatically found correspondences. In this second step, "Align Validation (2)", detailed in section 9.4.2, the expert refers to devices specifications to update and validate the matching. After the expert validation, the alignment file containing tuples and similarity values is transformed into an OWL ontology. In other words, the left and right entities are currently related with the *owl:equivalentTo* property. Then, the left, right and the alignment ontologies are merged into a single ontology containing the equivalent properties. The merged ontology contains the entities from both ontologies and the relations between.

The third step applies "Rules (3)" to automatically detect patterns between the equivalent correspondences. The rules detect compositions between actions. For example, an action on a device can be equivalent to the union of two other actions on another device. For each detected pattern between entities, the ontology is automatically annotated using composition relations relating the concerned entities.

Moreover, a matching between two actions is not enough to presume that they are compatible, their input/output parameters also need to be considered. Therefore, the patterns explained in section 9.4.3, detect compatible actions based on their input/output matched parameters.

The fourth step, "Alignment Adaptation (4)", is the final and optional step of the device matching. Actually, not all the correspondences between actions are simple and can be resolved only by linking the entities. The adaptation might needs data conversions and loops, for example, a temperature conversion, $°C = (5/9)(°F - 32)$. Thus, we overview in section 9.4.4, how the expert specifies an adaptation behavior using a high level API.

The valid ontology is then used as an input for the proxy generation and the device and service adaptation. We detail next the four major steps of the device matching.

9.4.1 Ontology Alignment

The ontology alignment step is applied on two ontologies O_1 and O_2 which abstract two devices description, for example the UPnP and DPWS Light ontologies shown in Figure 9.8. The ontology alignment uses the following matching strategy, shown in Figure 9.16, to automatically detect correspondences between equivalent services, actions and parameters.

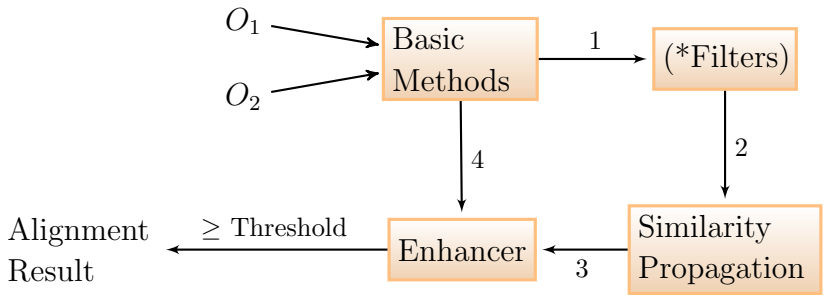


Figure 9.16: Aligner Simple description

The matching strategy consists of four steps, it takes two generated ontologies conformed to the meta model and detects correspondences between. We detail next each step of the matching.

9.4.1.1 Basic Methods

The first step of the alignment relies on the basic matching methods described in chapter 8. Thus, for each method k used, such as *SMOA, Hamming, Leveinstein*, we obtain an output similarity matrix Sim_Matrix_K shown in (9.1) between n (resp. m) entities of the ontology O_1 (resp. O_2).

$$Sim_Matrix_K = \begin{bmatrix} SimK_{1,1} & \cdots & SimK_{1,n} \\ \vdots & \ddots & \vdots \\ SimK_{m,1} & \cdots & SimK_{m,n} \end{bmatrix} \tag{9.1}$$

Figure 9.17, shows a part of a basic alignment result between the `BinaryLight` entity of the UPnP Light ontology O_1 and entities of the DPWS Light ontology O_2. The similarities showed corresponds to the first line of the similarity matrix. The alignment algorithm using the basic techniques ignores the predefined unified concepts in the ontology such as `UPnP Device`, `DPWS Service`.

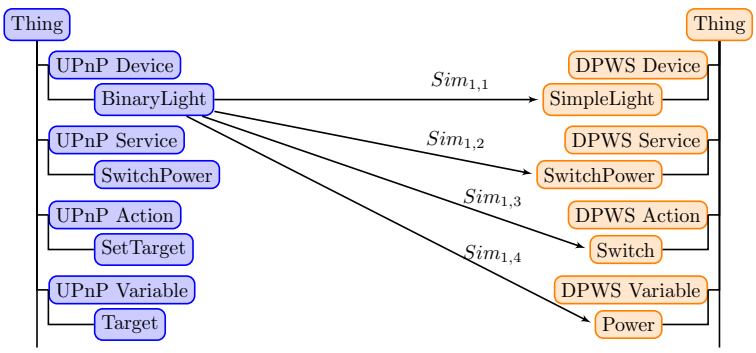


Figure 9.17: Part of a Basic Alignment Result Between Two Lights Ontology

Since each basic technique has its own advantages and drawbacks [Euzenat 07], we combine all these techniques and use a weight $\alpha_k \in \mathbb{R}^+[0,1]$ for each method k. Then, we apply an average weighed similarity between the concepts as shown in the equation 8.2 of chapter 8.

However, as mentioned in chapter 8, the string-based techniques can be used to detect relatively similar strings and mainly string extensions like a prefix or a suffix. Table 9.1 shows the similarity results returned by five string-based matching techniques. It is obvious that such techniques cannot detect antonyms (Up $\neq$ Down) or synonyms (Clock $\equiv$ Timer) and detect false matches between two similar strings (Particle, Article).

Method	δ^1(*"SetVolumeUp"*, *"SetVolumeDown"*)	δ(*"Clock"*, *"Time"*)	δ(*"Particle"*, *"Article"*)
1- Ngrams	0.7	0.0	0.9
1-Jaro	0.86	0.0	0.95
1-Hamming	0.69	0.0	0.0
1-Levenshtein	0.69	0.0	0.87
1-SMOA	0.88	0.0	0.96

[1] Similarity, $\delta = 1 - dissimilarity$

Table 9.1: Basic Matching Techniques Result with Synonyms and Antonyms

In order to improve the accuracy of the string-based techniques we propose the SMOA++ [Mora 10][3] basic matching technique inspired from SMOA [Stoilos 05] coupled with an external dictionary.

The SMOA matching technique relies on some patterns used by programmers when it comes to choose a descriptive names for their variables. The chosen names are usually a set of words trimmed and concatenated together. For example, the `SetVolumeUp` or the `SetVolumeDown` actions names. In the SMOA technique, the biggest common substring is searched between the two strings. Once found it is removed. Then, the search continues for the next biggest common substring between the two strings until none is found.

Definitely, in our context the device and service description contains trimmed and concatenated names. However, the SMOA technique is unable to detect synonyms, antonyms and cannot differentiate between two words semantically different having similar strings.

Thus, we propose the SMOA++ [Mora 10] also based on the substring search. The SMOA++ [Mora 10] consists of two major steps:

- Tokenization: The first step consists of separating the strings into tokens based on the WordNet dictionary. The SMOA++ searches for the biggest substring that can be identified in WordNet in each of the strings. Once the substring is identified, it is removed. Then, the search continues until no substrings can be identified. For example, the tokenization applied on the action name *CreateJob* returns the following three substrings based on WordNet: {Create, eat, Job}. However, since *eat* is a substring of *Create* then the token *eat* is not considered.

  The Tokenization is given more formally in the following definition:

  Definition 1 ($Tokenization_{SMOA++}$). *Let a string S composed of a set of substrings $\{s_1, s_2, \ldots, s_n\}$.*

  The $Common_{Sub}$ function between two strings returns the string having the biggest common substring. If no common substrings is found, the $Common_{Sub}$ returns both strings. The $Common_{Sub}$ can be defined as follow:

[3] SMOA++ is proposed by Felipe MORA during his final year project under my supervision

$$Common_{Sub}(s1, s2) = \begin{cases} \{s2\} \ if \ s1 \subset s2 \\ \{s1\} \ if \ s2 \subset s1 \\ \{s1, s2\} \ \ else. \end{cases}$$

The $List_{WordNet}(S)$ *function keeps only substrings of* S *which can be found in WordNet.*

$List_{WordNet}(S) \Rightarrow \{s_1, s_2, \ldots, s_m\}$, *where* $m \leq n$.

$Tokenization_{SMOA++}(S) = \{Common_{Sub}(\ List_{WordNet}(S)\ )\}$

For simplicity of illustration, we will consider for the next step the two string *SetVolumeUp* and *SetVolumeDown*. The tokenization of *SetVolumeUp* returns: *Set*, *Volume* and *Up* and the three substrings: *Set*, *Volume* and *Down* for the *SetVolumeDown*.

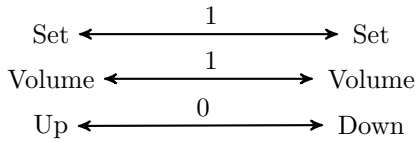


Figure 9.18: SMOA++ Matching

- The matching is the second step which consists first in finding the equal substrings using the *String Equality* technique. The *String Equality* similarity δ_{StrEqu} returns 0 if the strings are not identical and 1 if they are. As shown in Figure 9.18, the equivalent substrings will be assigned a similarity value of 1. We chose the string equality technique, however other basic techniques can also be applied.

  Then, we continue the matching for the non-equivalent strings. We query WordNet for their synonyms and antonyms. If the two strings are synonyms (Clock $\equiv$ Time) then we assign the similarity value $\delta_{Syn} = 1$ and if the they are antonyms (up $\neq$ down), then the whole antonym similarity δ_{Ant} is set to zero. The search for the antonyms is first applied on the two substrings, for example up and down. If no antonyms are found, we extend the search by retrieving the synonyms of each substring, i.e. the synonym list of *up* and the synonym list of *down*. Then we find antonyms between the two synonyms list. This allows to extend the antonyms search.

  The matching is given more formally in the following definition:

  Definition 2 (SMOA++ Matching)**.**

$$\delta_{StrEqu}(S_1, S_2) = \begin{cases} 1 \ if \ S1 = S2 \\ 0 \ \ else \end{cases}$$

$$\delta_{Syn}(S_1, S_2) = \begin{cases} 1 \ if \ S1 \ and \ S2 \ are \ Synonyms \\ 0 \ \ else \end{cases}$$

$$\delta_{Ant}(S_1, S_2) = \begin{cases} 0 \ if \ S1 \ and \ S2 \ are \ Antonyms \\ 1 \ \ else \end{cases}$$

Thus, with two strings composed of a set of substring $S_1 = \{s_1^1, s_2^1, \ldots, s_n^1\}$ and $S_2 = \{s_1^2, s_2^2, \ldots, s_m^2\}$ from the tokenization step. The SMOA++ similarity function is given by the following equation:

$$\delta_{SMOA_{++}}(S_1, S_2) = \frac{2 * \left(\sum_{i,j=1}^{n,m} \ \delta_{StrEqu}(s_i^1, s_j^2) \ + \ \delta_{Syn}(s_i^1, s_j^2)\right) * \prod_{i,j=1}^{n,m} \delta_{Ant}(s_i^1, s_j^2)}{length(S_1) + length(S_2)} \tag{9.2}$$

9.4.1.2 Filters

The result of the previous step, as shown in the equation 9.1, is a matrix holding $n*m$ similarity values between n concepts from the ontology O_1 and m concepts from the ontology O_2. The aim of the alignment is to find translation rules between equivalent types in order to replace for example an action invocation with another equivalent one.

Therefore, we apply two types of filters. The first keeps the best alignments between entities. For example, the filter only keeps the best similarity value between the four pairs of Figure 9.17. In other words, we apply a one-to-one cardinality mapping, this allows to pick up only the best pair of the matched entities between a concept of the ontology O_1 and the m concepts from O_2. Thus, the first filter is a maximum function which is applied on the matrix as follows:

$$Sim_Matrix' = max_{1 \leq k \leq m}(Sim_{k,1}, Sim_{k,2}, \ldots, Sim_{k,n}) \tag{9.3}$$

The similarity values of the Sim_Matrix', obtained after applying the first filter will be reused in the step 4 to improve the similarities.

The second filter is a "structural type" filter, we go through the alignment and keep only the correspondences between entities belonging to the same concepts types, (device-device), (service-service), (action-action) and (parameter-parameter).

The two following steps consist in enhancing similarity values based on the two ontologies structures.

9.4.1.3 Similarity Propagation

In this step, we apply a "down-top" similarity propagation on the ontology structure as shown in Figure 9.19. For example the similarity of the two services depends on the similarity of their matched actions if it is higher than a predefined threshold propagation value t_P. Furthermore, the actions' similarities is dependent on their input/output parameters' similarities. The function to propagate the similarity between two services S_1 and S_2, having n_{t_P} matched actions with a similarity value greater than a threshold t_P, is given by the following equation. We normalize the similarities which explains the division by n_{t_P} and 2:

if $\delta(S_1.Action_i, S_2.Action_i) \geq t_P / \ t_P \in \mathbb{R}^+[0, 1]$,

$$\delta_{Propagated}(S_1, S_2) = \frac{\delta(S_1, S_2) + (\ \sum_{i=1}^{n} \delta(S_1.Action_i, S_2.Action_i)\ /n_{t_P}\ )}{2}. \tag{9.4}$$

More generally, each similarity between two entities of a certain level (device, service, action or parameter) depends on its sub-levels similarities. In other words, the (device-device) similarity depends on the similarity of their matched services similarities. A service similarity depends on their actions' similarities and finally the actions similarity is dependent on their parameters' similarities.

9.4.1.4 Enhancer

This step is optional, it uses the similarity results of the first step before applying the "structural type" filter presented in section 9.4.1.2. The aim of this step is to propose alignments based on what is already detected.

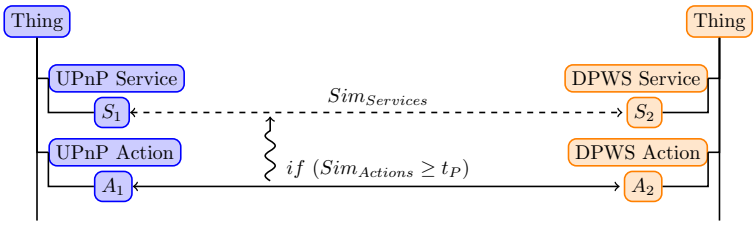


Figure 9.19: An illustration of the similarity propagation

For example, in Figure 9.20, a service S_2 has a high similarity with the action A_1 of the service S_1, and there is also a similarity between the two actions A_1 and A_2. However there is no alignment detected between the two services, S_1 and S_2.

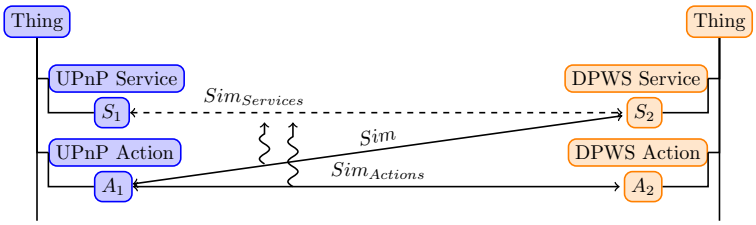


Figure 9.20: An illustration of the similarity enhancement

Since there is a match between the two actions (same-level) and between the actions and the service (cross-level), we can suppose that there is a possible relevance between the two services. Thus, we propose to enhance the similarity between the two services based on the similarities of the same and the cross levels if their values are equal or higher than a threshold t_E, i.e. between the actions pair and the service-action pair. We assign a confidence value α to the same-level similarity and the confidence value β to the cross-level similarity with, $\alpha,\ \beta \in \mathbb{R}^+[0,1]$ and $\alpha + \beta = 1$. Thus, the enhancer function is provided by the following equation:

if $\delta_{same-level}(A_1, A_2) \geq t$ and $\delta_{cross-level}(A_1, S_2) \geq t_E /\ t_E \in \mathbb{R}^+[0,1]$,

$$\delta_{Services_{Enhanced}}(S_1, S_2) = \frac{\delta_{Services}(S_1, S_2) + \alpha * \delta_{same-level}(A_1, A_2) + \beta * \delta_{cross-level}(A_1, S_2)}{2} \tag{9.5}$$

The equation 9.5 can be re-applied several times depending if S_1 and S_2 have multiple (action-action) and (service-action) pairs similarities. The *Enhancer* technique can also be applied on the three concepts types: device, service and action.

The output of alignment step is an alignment (.rdf) file containing multiple sets of tuples (leftEntity, Right-Entity, Similarity), see Listing B.3 in the appendix B.5. Each tuple represents an alignment between the left and the right entities of the two ontologies and the calculated similarity values. The expert loading the ontologies and triggering the alignment can choose a threshold value, then all similarity values higher than the threshold value are kept.

The alignments are based on heuristics, thus, an expert intervention is required to validate the found correspondences. We present next the alignment validation tool.

9.4.2 Expert Alignment Validation

Unfortunately, the ontology matching is still imperfect and heuristic based, even when adding a semantic dictionary such as WordNet, false matches can still be proposed as potential matches. Therefore, an expert intervention is needed to validate the mappings. The expert relies on the standard device specifications and manuals to validate the correspondences. Thus, the alignment validation is the second step which follows the ontology alignment step, as shown in Figure 9.21.

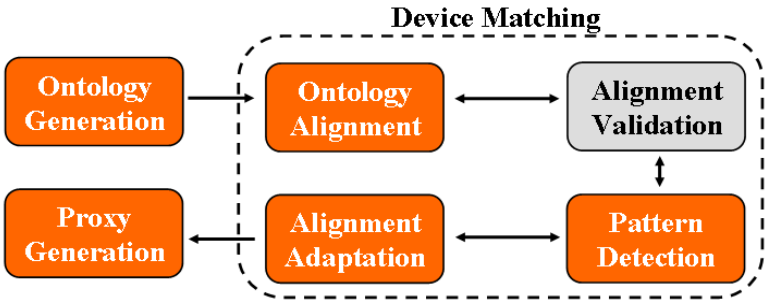


Figure 9.21: Step 2: Alignment Validation

As mentioned in chapter 8, multiple frameworks such as COMA++ [Aumueller 05], PROMPT [Noy 00] and ROMIE [Elbyed 09] propose ontology alignment techniques and an editing and validation GUI. However, these frameworks require an advanced knowledge in the ontology engineering domain in order to edit and correct the alignments.

In order to simplify the editing and adaptation we propose ATOPAI: an **A**lignment and anno**T**ation framew**O**rk for **P**lug and pl**A**y **I**nteroperability. It is based on the Alignment API [Euzenat 04] and provides a lot of useful features to simplify the alignment validation of the plug and play devices.

The ATOPAI framework is hosted on the operator platform, the ontologies can be statically generated by the *OWL Writers* then loaded in ATOPAI by the expert. The ontologies can also be uploaded by the *OWL Writers* deployed on the end users' sites upon a new device type detection. ATOPAI allows to load any ontology conformed to the predefined meta model. Each entity is annotated with its type (**D**:Device, **S**:Service, **A**:Action, etc) as shown in Figure 9.22. To calculate a mapping, the alignment method is chosen along with the trimming threshold t value (alignments above t are kept) and a semantic dictionary if needed.

Figure 9.22, shows an automatically calculated alignment between a UPnP and a DPWS light using SMOA [Stoilos 05]. The similarity value is shown on the line between the entities. The alignment result is expressed with an alignment format [Euzenat 04] with cells containing tuples (leftEntity, rightEntity, similarity). The result can be saved to an RDF file which can be loaded later. The expert can also saves the found tuples in a data base which can be queried to enhance a similarity value of a tuple if the saved value is higher then the current detected similarity.

The expert fixes the alignment by dragging lines between same type entities (device-device, service-service, etc). He can also add an alignment by only selecting two entities and using the Add Alignment button. ATOPAI assigns to the alignment added by the expert a similarity value set to 1 to avoid removing a valid alignment when trimming. To remove an alignment, the expert selects the line between the two entities and removes the alignment. All the alignments edited by the expert are shown in blue, the others are detected automatically. Figure 9.23 shows a part of the matching between two lights ontologies after the expert validation.

Once the mapping is validated, ATOPAI transforms the alignment format containing tuples (leftEntity, rightEntity, similarity) to an OWL Alignment ontology. The Listing B.3 in the appendix B.5 shows an alignment rdf result between a UPnP Binary Light and a DPWS Simple Light. The transformation of the alignment

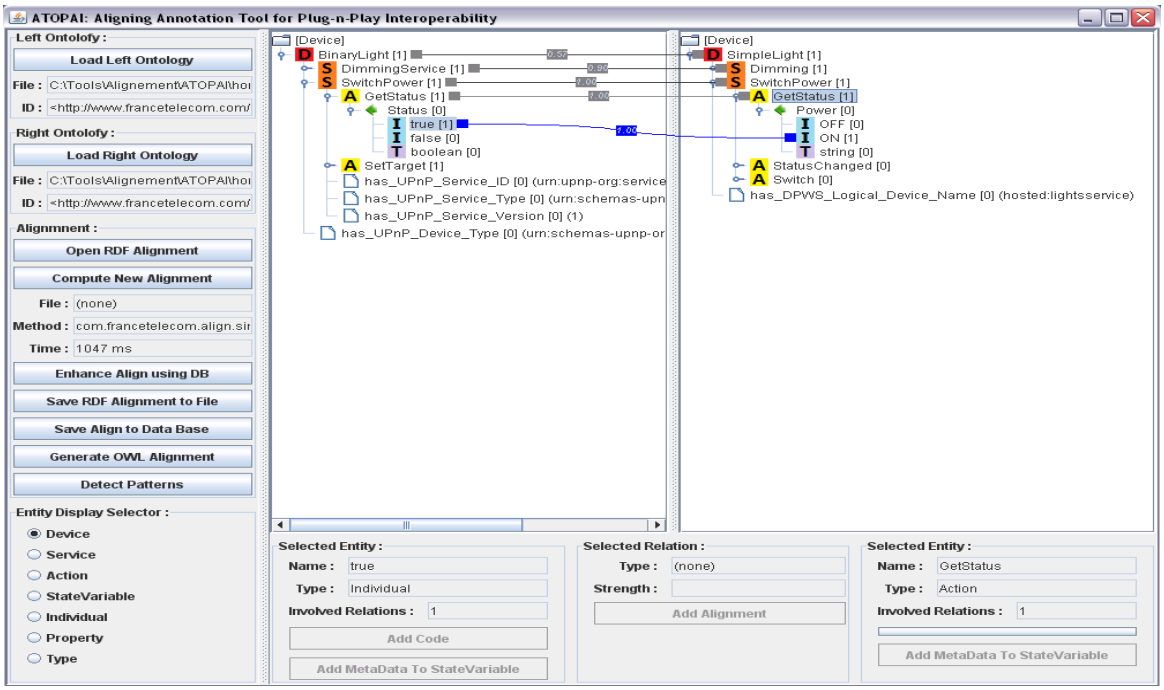


Figure 9.22: ATOPAI snapshot

format into an OWL ontology, translates the similarity values to the OWL object property (owl:equivalentClass) between entities. Then the three ontologies; the left, right and the align are merged into a single ontology in order to be used in the following steps.

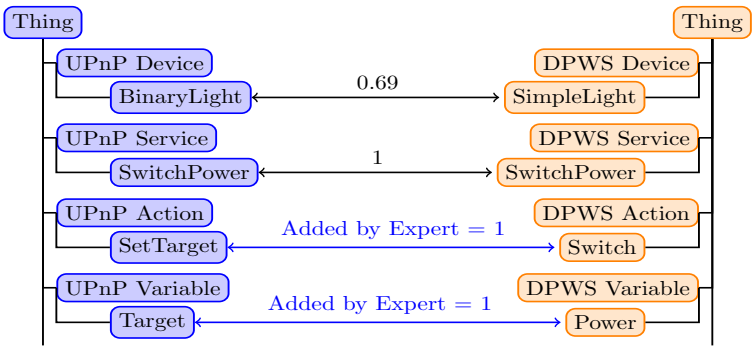


Figure 9.23: Part of the Alignment Result Between Two Lights Ontologies After Expert Validation

9.4.3 Pattern Detection

The matching between entities detects only one-to-one cardinality mapping as stated in section 9.4.1.2. In simple cases, we have one-to-one mapping actions, such as in Figure 9.23 where the `SetTarget` (boolean: true/false) is equivalent to the action `Switch` (String: ON/OFF). However, the adaptation between devices can require a composition adaptation. For example, on a device, an action can be equivalent to the composition of two or more other actions. Figure 9.25 shows an example of two fake clocks descriptions. The SMOA++ based alignment result detected only one of the correspondences, (SetClock $\equiv$ SetTime). During the validation, the expert added the alignment between the `SetClock` and the `SetDate` actions based on the two devices specifications. The alignment contains then a union composition, the UPnP action `SetClock` which takes two input parameters

(Date, Time) is equivalent to two actions `SetTime`(Time) and `SetDate`(Date). In other words, when the UPnP-DPWS Proxy receives the `SetClock` invocation, it must extract the values from both parameters Time and Date. Then, the UPnP-DPWS Proxy invokes on the DPWS Clock, the two actions `SetTime` and `SetDate`. Thus, the pattern detection step follows the alignment validation as shown in Figure 9.24.

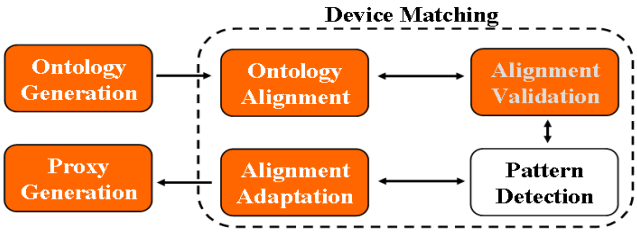


Figure 9.24: Step 3: Pattern Detection

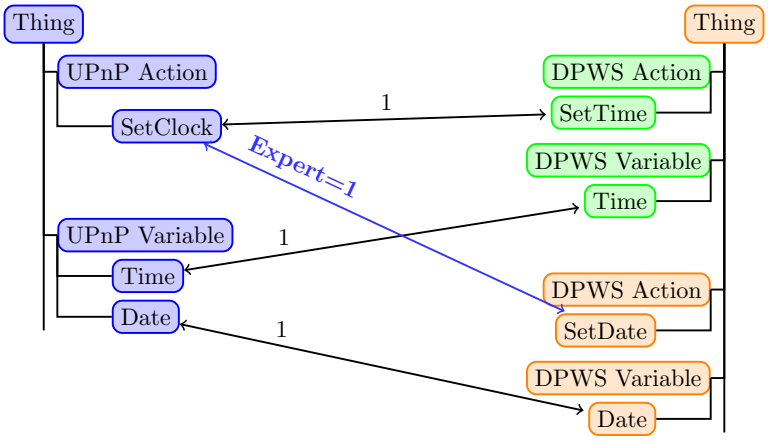


Figure 9.25: A Part of a SMOA++ Alignment between UPnP and DPWS Clocks

Another real example is also found on the standard UPnP and DPWS printers [UPnP , Microsoft 07], the UPnP action `CreateURIJob` with simple entries is equivalent to the association of two DPWS Actions, the `AddDocument` and the `CreatePrintJob` with complex structured entries, shown in Figure B.1 in the appendix B.2. A part of the mapping between UPnP.CreateURIJob and DPWS.CreatePrintJob shown in Table 9.2, reveals that the parameter `SourceURI` has an equivalent entry parameter `DocumentURL` for the DPWS action `AddDocument`. Consequently, the detected pattern is a UnionOf, CreateURIJob = UnionOf (CreatePrintJob, AddDocument). The `SendDocument` action can also be used instead of the `AddDocument` to send an attachment file instead of a URI.

In order to detect such compositions, we apply patterns on the previously expert-validated alignments. The pattern recognition as defined by Gonzalez [Gonzalez 78], is "a classification of input data via extraction of important features from a lot of noisy data". We detailed in chapter 5, how rules combined with basic logics can be used to extract information from what is already represented. In our approach, we use patterns to classify the matching between the actions: equivalent actions and complex compositions. The applied rules will detect patterns based on the alignment between entities. The rules will automatically annotate the ontology using specific properties between the actions to describe their composition type. Such composition information is exploited during the code generation in order to generate specific composition adaptation code between the affected actions.

However, annotating a composition type between actions is not enough to decide if the composition is valid,

their input/output parameters need to be considered. For simple devices, with actions having small number of input/output parameters, a manual checking of the compatibility can be carried out manually by an expert. However, Figure B.1 in the appendix B.2, shows the input parameter structure of the action `CreatePrintJob`. Therefore, when dealing with complex devices like a UPnP and a DPWS printers having a large number of parameters for each action, automating the validation and compatibility process decision become essential.

DPWS (CreatePrintJob)	UPnP (CreateURIJob)
PrintTicket/JobDescription/JobName	JobName
PrintTicket/JobDescription/JobOriginatingUserName	JobOriginatingUserName
PrintTicket/JobProcessing/Copies	Copies
X[1]	SourceURI
PrintTicket/DocumentProcessing/NumberUp/Sides	Sides
PrintTicket/DocumentProcessing/NumberUp/Orientation	OrientationRequested
PrintTicket/DocumentProcessing/MediaSizeName	MediaSize
PrintTicket/DocumentProcessing/MediaType	MediaType
PrintTicket/DocumentProcessing/NumberUp/PrintQuality	PrintQuality

[1] SourceURI has an equivalent variable DocumentURL in the DPWS action AddDocument

Table 9.2: Part of a Mapping between a standard DPWS and a UPnP printer action

Therefore, another set of decision rules are applied in order to detect valid matchings between actions compositions based on their parameters. This allows the expert to focus only on non valid ones in order to check if a potential adaptation or conversion can resolve the incompatibility. For example, the expert can make two actions valid by adding default values or writing high level adaptation code by referring to the specifications.

Thus, the patterns are used for three purposes. First to automatically annotate the ontology with union and sequential compositions of actions based on the expert validation. Second, to automatically propose to the expert non valid mappings in order to explore possible adaptation mechanisms such as setting default values and adding high level adaptation code. And the annotation will be exploited during the proxy generation in order to include in the proxy the adaptation behavior trough composition and redirection.

The pattern detection is an automatic operation triggered initially by the expert through ATOPAI once the expert's validation is accomplished.

We detail next, the three major categories of patterns used in our approach. The patterns are detected by applying rules on the ontology [El Kaed 11a].

9.4.3.1 The Patterns

In this section, we present the patterns used to automatically annotate the ontology with union and sequential compositions of actions based on the expert validation. We also provide the formal definition of each pattern in the first order logics along with the some rules expressed in the Ontology Pre-Processing Language [Šváb Zamazal 10] overviewed in chapter 5.

The following paragraph presents some general definitions and notations used in the definitions and the equations employed later in this section.

Definition 3 (General symbol definition)**.**

Consider O an ontology, A a set of actions of the ontology O and P a set of input/output parameters.

- $\forall f, g \in O,\ f \equiv g \iff (f,\ owl{:}equivalentTo,\ g)$

- $\forall f \in A,\ \forall x, y \in P_f,\ y = f(x) \iff f\ hasInput\ (x)\ and\ hasOutput\ (y)$

$y = f(\phi)$ is used if the action f has no input or the input is not considered. For simplicity of illustration, two equivalent input/output parameters will be represented with the same name but with the index of their action names. For example, x_f is a parameter of the action f.

1. ***Simple_Mapping*** is a pattern between two actions, having a one-to-one simple mapping. This property has the following three sub-properties:

   (a) ***Simple_Mapping_Input***: two actions are related with a Simple mapping input, if there is at least one matching between their input parameters. For example, Switch(Power) ≡ SetTarget(Target). More formally, the Simple_Mapping_Input definition can be specified using the first order logics as follows:

   Definition 4 (Simple_Mapping_Input)**.** $\forall f, g \in A,\ \exists x_f \in P_f,\ x_g \in P_g\ /$
   $f(x_f) \wedge g(x_g) \wedge (f \equiv g) \iff f\ Simple_Mapping_Input\ g$

   To detect the Simple_Mapping_Input pattern, we apply the following rule shown in Listing 9.1 written in the Ontology Pre-Processing Language [Šváb Zamazal 10] already overviewed in chapter 5. Once the rule is applied on the ontology, it will add a relation between the concepts (classes) of the ontology. For example, the rule will link the detected pair of entities *Switch* and *SetTarget* with the object property *Simple_Mapping_Input*. For conciseness, we detail one example written in the OPPL language per pattern category.

```
1 ?f:CLASS, ?g:CLASS, ?xf:CLASS, ?xg:CLASS
2 SELECT ?f subClassOf has_UPnP_Input some ?xf,
3 ?g subClassOf has_DPWS_Input some ?xg,
4 ?f equivalentTo ?g, ?xf equivalentTo ?xg,
5 BEGIN
6 ADD ?f subClassOf Simple_Mapping_Input some ?g
7 END;
```

   Listing 9.1: The OPPL rule applied to detect the Simple_Mapping_Input Pattern

   The first line of the Listing 9.1 declares four variables of type class. Line 2, selects any class related with another class in the ontology through the property *has_UPnP_Input*. Line 4, specifies that the selected classes f and g must also be related with an *equivalentTo* property. The same applies to the selected classes x_f and x_g. Line 6, annotates the ontology with the property *Simple_Mapping_Input* between the selected classes f and g verifying the previous criteria specified in the select part of the OPPL rule.

   In other words, the rule specified in the Listing 9.1, will select a class f (respectively a class g) having as *UPnP_Input* (respectively *DPWS_Input*) a class x_f (respectively a class x_g). Moreover, the selected classes f and g must be equivalent in the ontology. The classes x_f and x_g must also be equivalent. The found classes verifying the previous criteria will be annotated with the property *Simple_Mapping_Input*.

   (b) ***Simple_Mapping_Output***: two actions having at least one matching of output parameters. For example, (Power=GetStatus()) ≡ (Status=GetStatus()). More formally, the Simple_Mapping_Output definition can be specified as follows:

   Definition 5 (Simple_Mapping_Output)**.** $\forall f, g \in A,\ \exists y_f \in P_f,\ y_g \in P_g\ /$
   $(y_f = f(\phi)) \wedge (y_g = g(\phi)) \wedge (f \equiv g) \iff f\ Simple_Mapping_Output\ g$

(c) ***Simple_Mapping_Input_Output***: is the association of the two previous patterns. It is defined as follows:

Definition 6 (Simple_Mapping_Input_Output)**.** $\forall f, g \in A$,
$(f\ Simple_Mapping_Input\ g) \wedge (f\ Simple_Mapping_Output\ g) \Leftrightarrow f\ Simple_Mapping_Input_Output\ g$

2. ***Union_Mapping***: this pattern occurs when an action is equivalent to the composition of two or more actions with no predefined order. There is three categories of the union mapping. We rely on the previously simple detected patterns in order to identify the *Union_Mapping* pattern. For conciseness, we only define the Union_Mapping_Input, where the input parameters are considered. However, we also detect patterns for the Union_Mapping_Output based on the output parameters.

(a) **One-to-N**: At least three actions are involved. An example of this pattern is shown in Figure 9.25 where the *SetClock* is equivalent to the union of the two actions *SetDate* and *SetTime*.

$$SetClock(Time, Date)\ \textbf{\textit{Union_1_to_n}} \begin{cases} SetDate(Date) \\ SetTime(Time) \end{cases}$$

We provide in Definition 7, One-to-2 Union_Mapping, however the definition can be generalized to n actions.

Definition 7 (Union, One-to-N)**.** $\forall f, g, h \in A\ /$
$(f\ Simple_Mapping_Input\ g)\ \wedge\ (f\ Simple_Mapping_Input\ h) \Longleftrightarrow f\ Union_{Input} 1 \rightarrow n\ \{g, h\}$

The Union_1_to_n_Input pattern can be detected through the following OPPL rule shown in Listing 9.2. Applied on the ontology of Figure 9.25, the rule will add the property *Union_1_to_n* pattern between the three actions *SetClock*, *SetDate* and *SetTime*.

```
?f:CLASS, ?g:CLASS, ?h:CLASS
SELECT ?f subClassOf Simple_Mapping_Input some ?g,
?f subClassOf Simple_Mapping_Input some ?h,
WHERE ?f != ?g, ?g != ?h // f, g and h cannot be the same class
BEGIN
ADD ?f subClassOf Union_1_to_n_Input some ?g,
ADD ?f subClassOf Union_1_to_n_Input some ?h
END;
```

Listing 9.2: The OPPL rule applied to detect the Union_1_to_n_Input Pattern

The Union_1_to_n_Output pattern can also be detected based on its output parameters as shown in the following example. The action f is equivalent to the actions g and h along with their parameters y and z. The Union_1_to_n_Output rule is based on the *Simple_Mapping_Output*.

$$\{y_f, z_f\} = f(\phi)\ \textbf{\textit{Union_1_to_n_Output}} \begin{cases} y_g = g(\phi) \\ z_h = h(\phi) \end{cases}$$

The Union_1_to_n_Input_Output pattern is based on the two previously detected patterns.

(b) **N-to-M**: The *Union_n_to_m* patterns can be found on devices where two or more actions are equivalent to the union composition of multiple actions. Such union is detected based on their equivalent parameters as shown in Figure 9.26. The action f is a union of the actions h and k. However, to invoke the action h, the input parameter b_h is needed which can be retrieved only if the action g is invoked. Thus the *N-to-One* union mapping and the *N-to-M* represents a sort of a deadlock union.

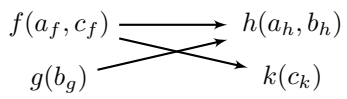


Figure 9.26: N-to-M Union Mapping

The two actions f and g must be both invoked to proceed with the adaptation and the invocation of the equivalent actions.

These two patterns make the service adaptation indeterministic. When the generated proxy receives an f invocation, it has two options. The first, is to wait for a lapse of time and adapt other received actions in the meanwhile until the proxy receives the invocation of the action g. Then, the proxy can invoke the actions h and k on the equivalent device.

The second option consists in the following, when the action f is received, the next received action invocation must be g in order to apply the adaptation and invoke h and k. If another action is received instead, the invocation redirection of h and k are ignored.

The **N-to-M** pattern in general cannot be adapted, since the invocation is unpredictable. The two overviewed options provoke undesirable behavior since the application might be expecting an immediate return value after an action invocation in order to proceed with a predefined ubiquitous task. In the first option, the application can wait indefinitely before receiving a return value or detecting the execution of the desired action in the ubiquitous environment. The second option, ignores the adaptation if the two actions are not invoked one after another. Thus, the proxy might not execute the desired task at all. Additionally, there is no guarantee that the application will invoke both actions f and g since the application might just need to execute the action f or g separately.

Despite of these drawbacks, when the *N-to-M* union patterns occur, the ontology is still annotated with the composition relation *Union_n_to_m*. However, its is up to the expert to decide whether to consider the detected union as valid or to ignore it. His decision is based on the device type and the dependency degree between the actions f and g. A high dependency between the two actions makes this pattern valid for composition and adaptation. Such high dependency between two actions is found for example on the DPWS standard printer where the action *CreatePrintJob* must be followed up by the invocation of the action *AddDocument* or *SendDocument* in order to accomplish the printing task[4].

Now consider the following, we chose the DPWS devices as a pivot instead of UPnP and we generate a DPWS proxy which exposes any UPnP printer as a DPWS printer and is able to interact with the printer. Thus, in this case the proxy can carry out the adaptation since the DPWS application interacting with the proxy will invoke the two DPWS actions on the proxy, the *CreatePrintJob* and the *AddDocument*. Thus, the proxy receiving both actions can extract the required values and invoke the equivalent UPnP action *CreateURIJob* on the UPnP Printer. Thus, this pattern can be valid, depending on the device and service specifications' and behavior.

Therefore, the detected patterns are annotated in the ontology and it is up to the expert to chose according to the device specifications and manuals if an adaptation can apply by setting default values or special configurations. We detail how the expert can carry out such adaptation operations using a high level language in section 9.4.4.

[4]Actually, there is a sequential union between the three actions. However, we only consider the union relation in this example in order to explain how a union adaptation can be carried out

(c) **N-to-One** is a specification of the *N-to-M* union, therefore, the same drawbacks and constraints apply.

3. ***Sequential_Union***: is a union mapping with a predefined order between actions. There is also three categories for a sequential union. In order to detect a sequential union, the rules rely on the previously union detected patterns. The only difference between a sequential and the previously detailed union pattern resides in the parameters. In a sequential union, an output parameter of an action is actually an input parameter of another action. Therefore, an invocation order is established between the two actions.

(a) **One-to-N**: At least three actions are involved in the *One-to-N* sequential union as shown in the following:

$$y_f = f(x_f) \quad \textbf{\textit{Sequential_1_to_n}} \begin{cases} (1) \quad r_g = g(x_g) \\ (2) \quad y_h = h(r_h) \end{cases}$$

The action f has an output parameter y_f and an input parameter x_f. The action f shares an equivalent input parameter with the action g. The action f also shares an output parameter with the action h. Thus, the action f is related with a *Union_1_to_n* property with the actions g and h. However, the action h has an input parameter r_h which is the output parameter of the action g. Therefore, the action adaptation requires first the invocation of the action g, then, waiting for its return value r_g. Furthermore, invoke the action h having as input the parameter r_h. And finally, return the y_h value.

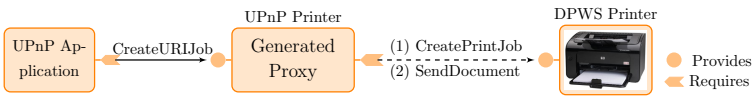


Figure 9.27: Sequential Union detected on the Standard Printers

We detected this pattern on the standard UPnP and DPWS Printer devices, *UPnP.CreateURIJob* Sequential_1_to_n (*DPWS.CreatePrintJob*, *DPWS.SendDocument*). The DPWS action *CreatePrintJob* and the DPWS action *SendDocument* share the *JobId* parameter which the output of the *CreatePrintJob* and is used as an input parameter by the *SendDocument*. As shown in Figure 9.27, upon receiving the *UPnP.CreateURIJob* action invocation, the proxy first retrieves the UPnP input parameters values, transforms them into the equivalent *DPWS.CreatePrintJob* parameters and invoke the first DPWS action. The proxy waits for the *JobId* returned by the printer as the output parameter of the *CreatePrintJob*. Once received, the proxy extracts it and use the same value as an entry of the *DPWS.SendDocument*. Table 9.2 shows a part of the input parameters used by the three actions. We detail in chapter 11 the mapping parameters between the three actions.

In order to detect the sequential union mappings, we first detect the union mapping then the sequential dependency (*has_Next*) between actions of the same device. *has_Next* is a binary relation defined as follows:

Definition 8 (has Next). $\forall f, h \in A, \quad f \ \textit{has_Next}\ h \iff$

$$\begin{cases} (1) \quad f \neq h \quad \text{and} \quad Output(f) \cap Input(h) \neq \emptyset \\ \qquad\qquad\quad \text{and} \quad Output(h) \cap Input(f) = \emptyset \\ \quad (or) \\ (2) \quad \exists\ g \in A\ /f\ \textit{has_Next}\ g \text{ and } g\ \textit{has_Next}\ h \end{cases}$$

We detect the *hasNext* pattern by applying the three following OPPL rules:

The rule shown in Listing 9.3 allows to detect the dependency between the actions based on the equivalence between the actions and the equivalence between their input and output parameters. Lines 2 and 3 allows to detect the unions and the lines 5, 6 and 7 selects the actions having inputs and outputs. And finally, lines 7 selects the actions with equivalent parameters.

The second rule is used to detect a cycle between two actions, (g has_Next h ∧ h has_Next g). In such a case, the expert validating the ontology is notified and the cycle is broken by removing the *has_Next* properties, as shown in Listing 9.4. The third category of rules is used to detect the transitive clause.

```
1 ?f:CLASS,?g:CLASS,?h:CLASS,?x:CLASS,?y:CLASS,?r:CLASS
2 SELECT ?f subClassOf Union_1_to_n some ?g,
3 ?f subClassOf Union_1_to_n some ?h,
4 ?f subClassOf has_UPnP_Output some ?x,
5 ?g subClassOf has_DPWS_Output some ?y,
6 ?h subClassOf has_DPWS_Input some ?r,
7 ?x equivalentTo ?y, ?x equivalentTo ?r
8 WHERE ?f != ?g, ?g != ?h
9 BEGIN ADD ?g subClassOf hasNext some ?h END;
```

Listing 9.3: An OPPL rule applied to detect the has_Next Pattern in general

```
?g:CLASS, ?h:CLASS
SELECT ?g subClassOf has_Next some ?h,
?h subClassOf has_Next some ?g
BEGIN
REMOVE ?g subClassOf has_Next some ?h,
REMOVE ?h subClassOf has_Next some ?g
END;
```

Listing 9.4: The OPPL Cycle detection Rule

Figure 9.28 shows a generalized complex mapping between an action f and a set of union and sequential actions. The *has_Next* relation is modeled with a **DAG** Directed Acyclic Graph: G since *has_Next* is a partial order relation: irreflexive, asymmetric and transitive. G is sorted to an ordered graph, as shown in Figure 9.29, using a topological sorting algorithm[Frank 66] as shown in Algorithm 1, the proxy can then handle and invoke the ordered action list properly.

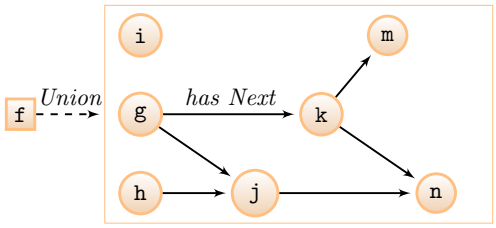


Figure 9.28: Example of a complex Sequential and Union Mappings

Figure 9.29: A possible order of the example in Figure 9.28

Finally, based on the *hasNext* pattern and the *Union_1_to_n* pattern. The Sequential_1_to_n definition can be specified more formally as follows:

Algorithm 1 Topological Sorting

List : list of actions

Ordered_List : list of ordered actions

(1) Find an action $f \in$ List with no incoming *has_Next* edge and add it to the Ordered_List

(2) Remove f from List

(3) Recursively compute a topological ordering of (List-1)

(3a) Append this after f in the Ordered_List

Definition 9 (Sequential Union 1 to n)**.** $\forall f, g, h \in A,\ /$
$f\ Sequential_1_to_n\ (g, h) \Longleftrightarrow (f\ Union_1_to_n\ g)\ \wedge (f\ Union_1_to_n\ h)\ \wedge (g\ hasNext\ h)$.

The following rule shown in Listing 9.5 is applied to detect the sequential union.

```
?f:CLASS,?g:CLASS,?h:CLASS,?x:CLASS,?y:CLASS,?r:CLASS
SELECT ?f subClassOf Union_1_to_n some ?g,
?f subClassOf Union_1_to_n some ?h,
?g subClassOf hasNext some ?h
WHERE ?f != ?g, ?g != ?h
BEGIN ADD ?f Sequential_1_to_n some ?h,
ADD ?f Sequential_1_to_n some ?g END;
```

Listing 9.5: An OPPL rule applied to detect a Sequential Union Pattern

(b) The **N-to-One** and the **N-to-M** sequential unions categories share the same drawbacks as the *N-to-M* unions.

$$\begin{pmatrix} (1)\ a_g = g(x_g) \\ (2)\ y_h = h(a_h) \end{pmatrix} \quad \textbf{\textit{Sequential_to_One}} \quad y_h = f(x_h)$$

The sequential union N-to-One in general is hard to adapt. Consider an application invoking $a_g = g(x_g)$, it expects a return value a_g. This value is then used as an input when invoking $y_h = h(a_h)$. Thus, the proxy carrying out the adaptation cannot generate a value of a unless it is well defined in a certain context or it is a random number.

However, if the parameter a is the *JobId* parameter returned by the DPWS standard action *CreatePrintJob*, then, in this case, the proxy can be configured to return any number and wait for the application to invoke the action h so it can return the y value.

Thus, the detected patterns are annotated in the ontology and it is up to the expert to choose according to the device specifications and manuals if an adaptation can apply by setting default values or special configurations. We detail how the expert can carry out such adaptation operations using a high level language in section 9.4.4.

In this section, we detailed how the patterns can be applied on the ontology in order to detect union and sequential compositions. However, a detected composition between two actions is not enough to presume that they are compatible, their input/output parameters equivalence and cardinality also need to be considered. Therefore, we detail in the next section how we detect compatible actions and point out the non valid ones for a potential adaptation.

9.4.3.2 The Matching Concepts

We provide in this section, the matching concepts which allow to automatically decide if a simple or a union mapping is valid or might be adapted by an expert. The matchings concepts are based on the equivalent

input/output parameters of the actions' pattern mapping type previously detected using rules. The detected pattern will provide information when specifying the matching concept and counting the number of equivalent parameters. For instance, a union mapping refers to more than one action, thus more parameters need to be taken into account.

In order to detect a valid mapping between two sets of actions, we take several criteria into account such as the number of equivalent input/output parameters between the actions. Moreover, an important criterion relies on the number of the satisfied entry parameters values required by an action. For example, consider the following *SequentialUnion_1_to_n* composition between the actions f, g and h:

$$c_f = f(a_f, b_f)\ \textbf{\textit{SequentialUnion_1_to_n}} \begin{cases} c_g = g(a_g) \\ d_h = h(c_h, b_h) \end{cases}$$

The mapping between the two sets of actions $\{f\}$ and $\{g, h\}$ is valid even if the number of parameters is not the same. The action f is not expecting a return value d therefore, the return value of the action h can be ignored by the proxy during the adaptation. Additionally, the parameter c_g is an output parameter of the action g and an input parameter of the action h, thus invoking the two actions in the correct sequential order will resolve the dependency. The input parameters a and b are equivalent in the both sets of actions, therefore the values can be transfered when the invocation of f is received.

The mapping validity on two sets of actions can be carried out manually on a small number of actions and parameters. However, on more complex compositions and devices where an action has several parameters, the manual check of the validity becomes a tedious task and error prone. Therefore, we provide the matching concepts based on *Paolucci*'s matching [Paolucci 02] detailed in section 7.1.1. *Paolucci* proposes four matching degrees which can be applied on a **common ontology** by exploiting the taxonomic structure between the concepts' semantics. In our work, the ontology contains only equivalent relations and the composition patterns.

We provide in the following some definitions used later to calculate the decidability of the actions validity.

Definition 10 (Matching Definitions)**.**

- $\forall a \in A,\ np_{Input}(a)$ is the number of parameters the action a has as input.
- $\forall\, a, b \in A,\ nbEqual_{Input}(a, b)$ is the number of equivalent input parameters between actions a and b.
- $\forall\, a, b \in A,\ np_{Common}(a \cap b) =$ number of common parameters between actions a and b.
- $\forall\ n \in \mathbb{N}^+,\ \forall a_1, a_2, ...a_n \in A,\ ,\ S_{a_n} : \{a_1, a_2, \ldots a_n\}$ is a set of actions.

In order to automatically detect valid compositions and mappings, we propose the *MConcept* equation. The $MConcept_{Input}$ takes two sets of actions S_{a_n}, S_{b_m}, for example $S_{a_n} = \{f\}$ and $S_{b_m} = \{g, h\}$ from the previous example. The $MConcept_{Input}$ returns an *MConcept* similarity value in $\mathbb{R}^+[0,1]$ which is calculated based on the equivalent input parameters divided by the number of the input parameters. There is also the $MConcept_{Output}$ which is based on the equivalence of the actions and their output parameters. For conciseness, we only detail the $MConcept_{Input}$ since the $MConcept_{Output}$ can be calculated using equivalent output equations. Thus, the matching concept is defined as follows:

Definition 11 (The Matching Concepts)**.** $\forall\, S_{a_n}, S_{b_m} /\, a_i, b_j \in A,\ n, m \in \mathbb{N}^{+*}$,

$$MConcept_{Input}(S_{a_n}, S_{b_m}) = \frac{nbEqual_{Input}(S_{a_n}, S_{b_m}) * 2}{Parameters_{Input}(S_{a_n}, S_{b_m})}$$

- $nbEqual_{Input}(S_{a_n}, S_{b_m})$ is the number of **equivalent** input parameters between the two sets of actions. It is calculated as follows: $\forall S_{a_n} : \{a_1, a_2, \ldots a_n\},\ S_{b_m} : \{b_1, b_2, \ldots b_m\}$,

  $nbEqual_{Input}(S_{a_n}, S_{b_m}) = \sum_{i=1}^{n} \sum_{j=1}^{m} (nbEqual_{Input}(a_i, b_j))$.

- $Parameters_{Input}(S_{a_n}, S_{b_m})$ is the number of input parameters between the two sets of actions. It is based on the number of input parameters of each action. However, we count only once the common parameters shared between sequential actions. The $Parameters_{Input}(S_{a_n}, S_{b_m})$ is calculated as follows:

  $Parameters_{Input}(S_{a_n}, S_{b_m}) = np_{Input}(S_{a_n}) + np_{Input}(S_{b_m}) - np_{Common}(S_{a_n}) - np_{Common}(S_{b_m})$.

- $np_{Input}(S_{a_n}) = \sum_{i=1}^{n} np_{Input}(a_i)$.

- $np_{Common}(S_{a_n}) = np_{Common}(\cap_{i=1}^{n} a_i)$.

If S_{a_n} and S_{b_m} have no input parameters and only outputs, then the $MConcept_{Input}(S_{a_n}, S_{b_m}) = 1$. The $MConcept_{Input}$ between the two sets of actions $\{f\}$ and $\{g, h\}$ returns the following similarity value $MConcept_{Input} = \frac{2*2}{2+3-1} = 1$. There is two equivalent input parameters between the two sets of actions. $\{f\}$ has two input parameters while $\{g, h\}$ have three input parameters and one common parameter c between the two actions g and h.

We extended Paolucci's[Paolucci 02] four matching degrees between matched services: Exact, PlugIn, Subsumes and Fail. *Paolucci* applied his proposed matching degrees on concepts belonging to the **same ontology** to detect compatibilities between services. *Paolucci* uses a reasoner to determine the compatibility between concepts based on a hierarchical classification in the common ontology.

In our approach, we only have the equivalent and compositions relations provided by the alignment and the patterns, therefore we redefine the following matching degrees between two sets of actions S_{a_n}, S_{b_m}, as follows:

- $ExactMatch_{Input}(S_{a_n}, S_{b_m})$: for **each** input parameter of S_{a_n} there is an *equivalentTo* relation with **each** input parameter of S_{b_m}, unless if the sequential union parameters resolve the non existing equivalence relations. For example, f(**x**,**y**) and g(**x**,**y**) have an $ExactMatch_{Input}$. The sequential union example between $\{f\}$ and $\{g, h\}$ is another $ExactMatch_{Input}$ example where the sequential union parameters resolve non-existing equivalence relations[5].

  The *ExactMatch* applies if $MConcept_{Input}(S_{a_n}, S_{b_m}) = 1$.

- $PlugIn_{Input}(S_{a_n}, S_{b_m})$: for **each** input parameter of S_{b_m} there is an *equivalentTo* relation with **some** input parameters of S_{a_n}. For example f(**x**,**y**,z) ≡ g(**x**,**y**). The parameter **z** can be ignored during the invocation since it has no equivalence on the action g. The *PlugIn* applies *iff* the following three conditions are valid:

  - $MConcept_{Input}(S_{a_n}, S_{b_m}) \neq 1$
  - $np_{Input}(S_{b_m}) = nbEqual_{Input}(S_{a_n}, S_{b_m})$
  - $np_{Input}(S_{a_n}) > np_{Input}(S_{b_m})$

- $Subsume_{Input}(S_{a_n}, S_{b_m})$: for **each** input parameter of S_{a_n} there is an *equivalentTo* relation with **some** input parameters of S_{b_m}. For example, f(**x**,**y**) ≡ g(**x**,**y**,z). The *Subsume* matching degree do not guarantee a successful translation between actions since some values of S_{b_m} are missing. The parameter **z** cannot be ignored, **g** is expecting a value. Therefore, it is up to the expert validating the alignment to verify the specifications and check if the parameter **z** can have a default value or other adaptation operations using ATOPAI. The *Subsume* applies *iff* the following three conditions are valid:

[5]In the following matching degrees' definitions the sequential union parameters resolving the non existing equivalence relations is considered. However, it is only removed from the definition for more clarity in the definition presentation

- $MConcept_{Input}(S_{a_n}, S_{b_m}) \neq 1$
- $np_{Input}(S_{a_n}) = nbEqual_{Input}(S_{a_n}, S_{b_m})$
- $np_{Input}(S_{a_n}) < np_{Input}(S_{b_m})$

- $Unknown_{Input}(S_{a_n}, S_{b_m})$: for **some** input parameters of S_{a_n} there is an *equivalentTo* relation with **some** parameters of S_{b_m} and does not verify any previously defined matching concept. For example, f(**x**,y,z) ≡ g(**x**,b,c).

(S_{a_n}, S_{b_m})	$Exact_{In}$	$PlugIn_{In}$	$Subsume_{In}$	$Unknown_{In}$
$Exact_{Out}$	✓	✓	?	?
$PlugIn_{Out}$	x or ?	x or ?	x or ?	x or ?
$Subsume_{Out}$	✓	✓	?	?
$Unknown_{Out}$	x or ?	x or ?	x or ?	x or ?

Table 9.3: Decision table, (✓:success, x:fail, ?:undefined)

Once the $MConcept_{Input}$ and the $MConcept_{Output}$ are calculated, we define the Table 9.3 to decide if the mapping between actions is a success (the translation between actions is satisfied), a failure or an undefined. We detail next, some examples to show how the table decisions are found.

Consider an $Exact_{Input}/Exact_{Output}$ matching, such matching is clearly valid and don't need an adaptation, thus the result is always valid. Consider the following mapping, (y=f(x,y,z)) ≡ (y=g(x,y)) which clearly represents a $PlugIn_{In}$ and an $Exact_{Out}$ mappings. The parameter z is not mapped and is not needed to invoke the action g, thus it can be ignored by the proxy during the adaptation, therefore the mapping decision is always valid. Now consider another $Exact_In/PlugIn_Out$ mapping, (y=f(x,y)) ≡ (g(x,y)), such mapping cannot be automatically classified as valid. The application invoking the action f on the proxy, is expecting a return value y, thus it is up to the expert to check if an adaptation is possible based on the returned value.

Table 9.4, shows the decision returned to the expert by ATOPAI between the DPWS and UPnP Printers. The first two actions are successful, the expert then focuses only on the undefined decisions to check if it can be turned to success. The *CreateURIJob* mapping, see Table 9.4, is turned to success by setting the *LastDocument* to true as a default value of the action *AddDocument*. The *GetPrinterAttributes* in Table 9.4 is turned to success by setting the *PrinterStatus* as default input of the action *GetPrinterElements*. All the successful mappings are used for the code generation while the failures are ignored.

The successful mappings are directly used in the code generation without any expert intervention. The undefined decisions require an intervention from the expert which checks if an adaptation is possible by providing default values set using ATOPAI. The adaptation is applied using a high level API to specify the behavior as detailed in the next section 9.4.4. The failure/undefined decision (x/?) is harder to adapt since the adaptation depends on the output or input parameters not having an equivalent relation. Thus as mentioned in section 9.4.3.1, with *Sequential_Union_n_to_m*, such adaptation is in general impossible and depends on the actions specifications and descriptions.

The adaption of the undefined decisions can be accomplished either by setting default values or by using a high level API to specify the adaptation behavior as detailed in the next section.

9.4.4 Expert Code Annotation

Rules automatically detects patterns, however, not all the correspondences between actions are simple and can be resolved only by linking the entities or setting default values. The adaptation might need data conversions

UPnP Action (S_{a_1})	DPWS Actions (S_{b_m}, m = 1,2)	Matching	Decision
CancelJob	CancelJob	$Exact_{Input}$, $Exact_{Output}$	✓
GetJobAttributes	GetJobElements	$Exact_{Input}$, $Subsume_{Output}$	✓
CreateURIJob	Seq.[1](CreatePrintJob, AddDocument)	$Subsume_{Input}$, $Exact_{Output}$	?
GetPrinterAttributes	$\cup$[2](GetPrinterElements, GetActiveJobs)	$Susume_{Input}$, $Subsume_{Output}$	?

[1] Sequential_1_to_n

[2] Union_1_to_n

Table 9.4: Equivalent actions for UPnP-DPWS Standard Printers

and loops, for example a temperature conversion between C degree and F using the following equation: °C = (5/9)(°F − 32). This step is optional and depends on the previous matching concepts results, as shown in Figure 9.30.

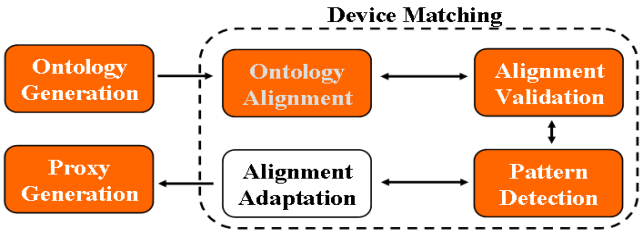


Figure 9.30: Step 4: Code Annotation

We detail in the following paragraph two use cases where the alignment is insufficient and adding code is necessary. The first use case requires a code adaptation while the second additionally requires calling an external service. Since, the UPnP and DPWS APIs require an advanced knowledge in both protocols and the base drivers API, we offer the expert a simple high level `Adaptation API` to add the conversion operations which are injected in the templates in specific call points during the code generation.

To add the adaptation code, the expert selects an action or state variable entity on the left ontology, the uses `Add Code Button` which allows to write the conversion behavior.

- **Use Case #1,** Consider two TV devices. TV_1 has a service `VolumeManager` with one action a_1:`SetVolume (int newVol)` used to set the new volume value. The TV_2 has a `VolumeManager` service with three following actions:

$$(a_1)\ SetVolume(newVol)\ ? \begin{cases} (a_2)\ currentVolume = GetVolume() : \text{retrieves the current value.} \\ (a_3)\ VolumeUp() : \text{to increase the volume value by 1.} \\ (a_4)\ VolumeDown() : \text{to decrease the volume value by 1.} \end{cases}$$

The action a_1 has no direct equivalence with the actions a_2, a_3 and a_4, however the following adaptation is possible. In our approach, the generated proxy is exposed as a device TV_1 and interacts with a device TV_2 as shown in Figure 9.31.

Listing 9.6 shows the adaptation behavior involving the three actions a_2, a_3 and a_4. When the proxy receives the `SetVolume` invocation it retrieves the `newVol` value, (line 5) then it invokes the `GetVolume` and retrieves the current volume level `currentVolume` on the TV_2, (lines [6,7]). The proxy calculates the difference between the two volumes' values, the `newVol` and the `currentVolume` on the TV_2. If the

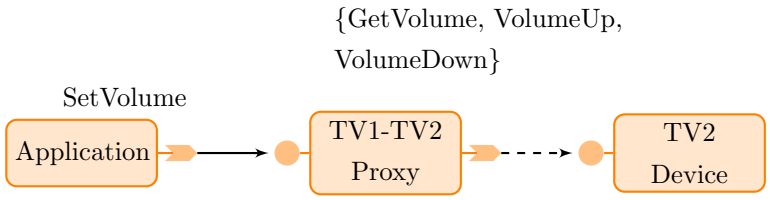


Figure 9.31: Adaptation Behavior: Use Case #1

difference is positive then the $newVolTV_1 > currentVolumeTV_2$, the proxy should increment the volume of the TV_2, (line 9). Else if the difference is negative then the proxy should decrement the volume of the TV_2, (line 10).

In order to include the adaptation behavior, the expert adds an alignment between the action a_1 and the three actions a_2, a_3 and a_4. Then the expert selects the `SetVolume` entity on the left ontology and adds the Code. When the `SetVolume` is invoked on the proxy, it behaves according to the following code.

```
1  //Select the right entities involved
2 DPWSAction a2 = getRightAction("GetVolume");
3 DPWSAction a3 = getRightAction("VolumeUp");
4 DPWSAction a4 = getRightAction("VolumeDown");
5  //Retrieve the new volume value, "this" refers to the left action
6 int v1 = (Integer) this.getInputValue("newVol") ;
7 a2.invoke(); // Invoke to retrieve the current volume on TV2
8 int diff = v1 -(Integer) a2.getRetValue("currentVolume");
9  //Increment
10 if(diff>0) { for(int j=0; j<diff; j++) a3.invoke();}
11  //Decrement
12 else{ for(int j=0; j<-diff; j++) a4.invoke();}
```

Listing 9.6: A High Level Adaptation Code

- **Use Case: #2,** Consider two devices D_1, D_2. The device $D1$ has an action: a_1:`SetPs(File ps)` which allows it to receive a *ps* format files. The device $D2$ provides an action a_2:`SetPdf(File pdf)` which allows it to receive a *pdf* format files.

  In this use case, the adaptation requires transforming the *ps* type files into *pdf* files. This operation is already implemented and provided by several services. Therefore, the adaptation only requires calling an ps-to-pdf format converter as shown in Figure 9.32. Thus, the expert adding the adaptation behavior of the proxy shown in Listing 9.7, specifies using the high level API to search and bind with an external service using the keyword *ExtService*. When the $a1$ is invoked on the proxy, it will invoke the method "File Convert(File ps)", of the external service "conv.ps2pdf.FileConvert" on the OSGi platform to convert a file. Once the file is converted, the proxy invokes the a_2`:SetPDF` on the D_2 using the *pdf* file format.

The added behavior is saved to a file and its reference is attached to the ontology. The added code will be injected in the next step during the proxy generation.

This step returns an ontology holding the equivalent parameters, actions and composition patterns between two devices. It can also include an adaptation behavior. Thus, this ontology captures in a high level language the adaptation and transformation rules to go from one device to another. The next step consists in going from the high level ontology language to the lower level execution code through code generation techniques. The generated code represents the proxy which provides the interoperability between the semantically equivalent devices.

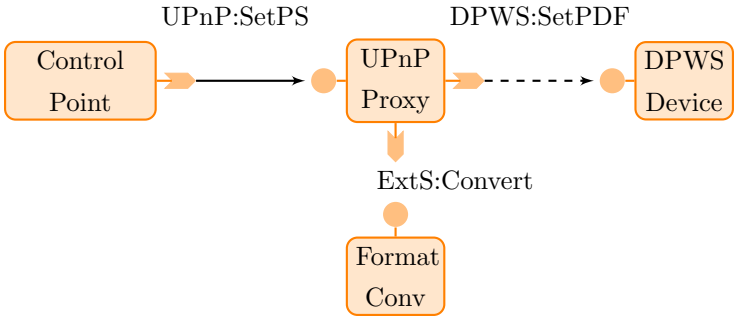


Figure 9.32: Adaptation through External Service Invocation: Use Case #2

```
1 import java.io.File;
2 DPWSAction a2 = getRightAction("SetPdf");
3 // Bind to the OSGi service "conv.ps2pdf.FileConvert"
4 ExtService s1 = new ExtService("conv.ps2pdf.FileConvert");
5 // Search for the method "Convert"
6 s1.setMethodtoInvoke("Convert");
7 // Set the input value
8 s1.setInputValue("File", (File) a1.getInputValue("ps"));
9 // Invoke the method "Convert"
10 s1.invoke();
11 // Retrieve the conversion result and invoke the a2
12 a2.setInputValue("pdf", (File) s1.getRetValue());
13 a2.invoke();
```

Listing 9.7: Adaptation through External Service Invocation

In this section, we detailed the device matching process with its four steps in order to semi-automatically detect correspondences between equivalent devices and services. The device matching allows to find equivalent entities and compositions between actions to achieve a successful substitution and interoperability between the services. Furthermore, since the matching is heuristics-based, we provide a simplified step to achieve an expert validation. Then, based on his validation, we apply rules to detect patterns between several entities. However, since not all the correspondences can be resolved automatically, we offer the expert an adaptation possibility by injecting an adaptation behavior during the code generation.

9.5 Concluding Remarks

In this chapter, we detailed our approach to resolve the heterogeneity between two equivalent devices. The approach combines two domains to achieve an interoperability between plug and play devices, the ontology alignment and the model driven engineering. The ontology alignment techniques based on heuristics, semi-automatically detect correspondences between two equivalent devices. Then, rules are applied to annotate the ontology with composition patterns and to report to the expert the non valid action compositions. Thus, the expert checks the device specifications to verify if an adaptation behavior can validate the detected compositions.

The validated ontology alignment contains entities of each device and the adaptation behavior to go from one device to another. Such adaptation behavior is represented in a high level language, the ontology language, independently from the implementation technical details.

Thus, we rely on the Model Driven Engineering techniques and exploit the high level representation of

the devices and the relations between to generate a proxy which transparently reconciliate the heterogeneity between the two devices. Thus, the generated proxy will expose itself as a specific UPnP device on the network. The UPnP applications will interact with the proxy transparently as a UPnP device. The received invocations from the UPnP applications are redirected to the real non-UPnP equivalent device on the network.

The next chapter details the implementation of the three modules *OWL Writers*, *ATOPAI* and *DOXEN*, then presents the experiments carried out on devices of the digital home.

Chapter 10

Implementation

"A good idea is about ten percent and implementation and hard work, and luck is 90 percent."

– Guy Kawasaki

Contents

10.1 OWL Writers **155**

10.2 ATOPAI **159**

10.3 DOXEN **167**

10.4 Experimentations **175**

We present in this chapter the implementation of our proposed approach. The implemented prototype performs a device to device adaptation based on three modules: the OWL Writers depicted in section 9.3.1, the Device Matching presented in section 9.4 and carried out by the ATOPAI framework and finally DOXEN overviewed in section 9.3.3 which generates proxies based on the ontology alignments.

The reminder of this chapter is structured as follows. First, we present the UPnP and DPWS OWL Writers' implementation details. Then, we suggest a Bonjour OWL Writer architecture since Bonjour is used only for discovery. Second, we present the ATOPAI Framework which allows to perform the device matching and offers a Graphical User Interface to allow alignment validation and code adaptation. Third, we present implementation details regarding DOXEN and the generated proxies. Finally, we provide the realized experimentations with the various plug and play devices.

10.1 OWL Writers

In our prototype, an OWL Writer module is an OSGi Bundle deployed on a Felix Apache OSGi Framework [Apache a]. Each specific OWL Writer subscribes to receive devices' announcements on the OSGi Framework. When a real Plug and Play device appears on the network, it is reified by a Base Driver, then the framework notifies the OWL Writer about the device arrival. Each OWL Writer relies on the Base Driver API to retrieve the device description and its supported capabilities. In our prototype, we used two base drivers, the UPnP Apache Base Driver [Apache b] and the SOA4D Base Driver [SOA4D b] supported mainly by Schneider Electric.

```
1 public void start(BundleContext Context) throws Exception {
2 this.ctxt = Context;
3
4 //Active and passive discovery of All UPnP Devices
5 ServiceReference[] sr = ctxt.getServiceReferences(UPnPDevice.class.getName(), null);
6
7 //UPnP LDAP like Filter
8 String UPnP_filter = "("+Constants.OBJECTCLASS + "=org.osgi.service.upnp.UPnPDevice)";
9
10 //register a Service Listener for UPnP Devices, will be notified when a new device appears
11 ctxt.addServiceListener(this, UPnP_Filter);
12
13 //get all UPnP devices available on the network
14   for (int i = 0; sr!=null && i < sr.length; i++) {
15    Object device = ctxt.getService(sr[i]);
16    if(ontology_Not_Yet_Generated((UPnPDevice) device)){
17         OB = new OWLBuilder((UPnPDevice) Device); //Build the ontology
18         //add OB.owlFile to the repository of ontology files...}} }
```

Listing 10.1: A UPnP OWL Writer Subscribes to UPnP Devices

An OWL Writer uses the OWL API 3.0 [OWL] to automatically generate an ontology based on the device description. The generated ontologies represent the device, the services, the actions and their variables. Each generated entity is a subclass of a well defined concept from the meta-model defined in section 9.3.1. In the lights ontology for example, *BinaryLight* (resp. *SimpleLight*) is a subclass of *UPnP_Device* (resp. *DPWS_Device*). The same applies on the rest of the entities which are also connected through object and data properties. For example, the device and the service are linked together with an objectProperty as follows: *BinaryLight has_UPnP_Service* some *SwitchPower*. The individuals in the ontology represents allowed values or instances of the variables, for example, the tokens ON/OFF are individuals (instances) of the variable *Power*.

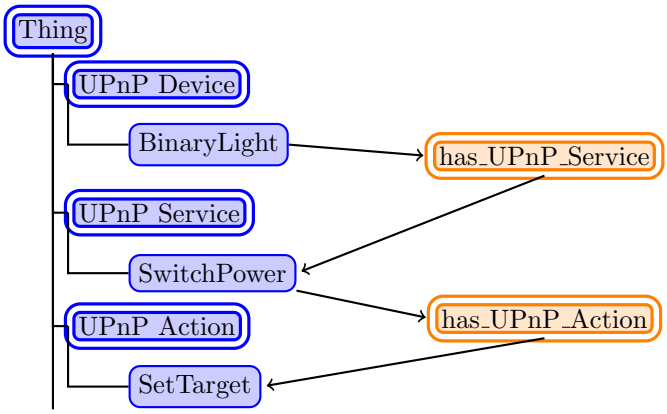


Figure 10.1: Generating Ontologies

Figure 10.1 shows a part of the generation. Each OWL Writer first loads a template ontology containing the main concepts and properties of the ontology. An example of the main concepts are represented in Figure 10.1 with double frames concepts, such as the UPnP Device, UPnP Service. Then, the OWL Writers goes through the device description and services and fills the ontology by placing the elements according to their type. For example, the *BinaryLight* is a device type, thus it will be represented in the ontology as a sub-concept of the ontology class UPnP Device. The same applies for the rest of the entities, service, action and variable. Additionally, for each sub-concept added, the OWL Writer, connects it to a specific property. For example,

when the *SwitchPower* service is added to the ontology, it is connected to the *BinaryLight* class with the OWL property *has_UPnP_Service*. Additional information are also attached to the ontology such as the version number and the complete UPnP Type (*urn:schemas-upnp-org-service:SwitchPower*), Version (*1*) and Id (*urn:upnp-org-servideId:SwitchPower:1*) used later by the generated proxy during description announcement.

Listing 10.2 shows a part of the UPnP OWL Writer service generation. The service name, version and type are extracted using the UPnP API. The Listing shows how the service is attached to its type and then related to its device. The *owl_upnpService* refers to the current service entity added to the ontology. Listing B.2 in the appendix B.4, shows the complete generated ontology of the UPnP Binary Light in the OWL syntax.

```
1 public void Write_Service(UPnPService uPnPService) throws OWLOntologyStorageException{
2  //owl_upnpService SubClass of owl_Service, SwitchPower subClass of Service
3  owl_upnpService = factory.getOWLClass(IRI.create(ontologyIRI +"#"+ Service_Name));
4  OWLAxiom axiom = factory.getOWLSubClassOfAxiom(owl_upnpService, Service);
5  manager.addAxiom(ontology, axiom); //add each axiom to the ontology
6
7  //Write Service Type
8  has_info=factory.getOWLDataProperty(IRI.create(ontologyIRI+"#has_UPnP_Service_Type"));
9  hasvalue=factory.getOWLDataHasValue(has_info,factory.getOWLStringLiteral(uPnPService.
       getType()));
10 axiom = factory.getOWLSubClassOfAxiom(owl_upnpService, hasvalue);
11 //Write Service Version and Service ID ...
12
13 //Connect upnpDevice with owl_upnpService, BinaryLight hasService SwitchPower
14 OWLClassExpression hasServiceSomeService=factory.getOWLObjectSomeValuesFrom(hasService,
       owl_upnpService);
15 OWLSubClassOfAxiom Subaxiom=factory.getOWLSubClassOfAxiom(owl_upnpDevice,
       hasServiceSomeService);
16
17 //Now add the actions, the service reference is passed to attach the actions to it
18  Write_Actions(uPnPService); ...}
```

Listing 10.2: UPnP OWL Writer Service Generation

The DPWS OWL Writer applies the same mechanism to generate ontologies, except for the operations and the messages' content. Such information cannot be retrieved using the DPWS Base Driver API, thus the DPWS OWL Writer retrieves the WSDL file from the real device. Then, the OWL Writer parses it and generates the operations names, variables and instances then attaches them to the ontology. The OWL Writers also support a shell command to generate ontologies by passing a description file (UPnP-XML or a WSDL). The next subsection, gives more information about the WSDL structure construction.

10.1.1 Flattening the WSDL

DPWS services' descriptions are exposed in a WSDL with hierarchical structures, as shown in Figure 4.3 and Figure B.1 in the appendix B.2. The *JobDescription* element, for example, is used by the following operations : *CreatePrintJob*, *AddDocument*, *SendDocument* and is located at level 3 or 4 from the root element depending on the entry of each action.

DPWS devices receiving invocations, expect to find the same structure when an action is received. Therefore, the same structure must be used when the proxy has to interact with the real device. In order to keep the structure and gain time during the code generation phase, we chose to flatten the parameters during the automatic generation of ontologies. Thus, the *JobName* element, in Figure 4.3, as an example, is represented by a path preserving the whole element structure as follows for each of the actions: Cre-

atePrintJob: *CreatePrintJobRequest/PrintTicket/JobDescription/JobName*, SendDocument: *SendDocumentRequest/.../JobDescription/JobName*, AddDocument: *AddDocumentRequest/.../JobDescription/JobName*).

The drawbacks of this solution are in the ontology construction and the aligning process since time is spent on the structure flattening by browsing the hierarchy. Additionally, instead of aligning once the structure, *./../JobDescription/JobName*, the *JobName* will be matched several times since the root of the structure is not the same.

However, One of the advantages of our choice is the clarity during the expert validation, parameters are now flattened and directly visible in the GUI/ATOPAI. Thus, the expert don't need to explore the hierarchy 4 or 5 level to validate an alignment. Second, it allows to gain time during the code generation since the hierarchy is represented as a path in the name, therefore, no need to explore the structure.

Consequently, the generation of structures is much more faster during proxy generation (less than 3 seconds for a printer proxy). Since the construction is relatively fast and the alignment is performed off line, then, loosing time during these two steps is acceptable in favor of gaining time during the dynamic code generation.

In chapter 4, we mentioned that the IGRS protocol relies on similar UPnP stacks and exposes its description using a WSDL. Thus, an IGRS OWL Writer will have the same implementation in order to go through the WSDL and build the ontologies automatically. Currently, there is no IGRS Base Driver supported on OSGi. Therefore, we did not develop an IGRS OWL Writer.

We detail next a possible solution for the Bonjour devices.

10.1.2 The "*Bonjour*" Exception

Devices hosting Bonjour use multicast-DNS[1] for the announcement and the discovery on the network as shown in Tables 4.4 and 4.3. For example, *"Apple LaserWriter 8500._ipp._tcp.local. Port 631"* is a bonjour printer announcement conformed to the following format *DeviceName._protocol._transportProtocol.Domain port number*.

Bonjour is only used to announce few information compared with the other plug and play protocols. In this example, the *ipp* protocol is used as an interaction protocol with this printer. Thus, other protocols are used on top of Bonjour for the interaction. For example, the Media Server *iTunes*(v4.2) uses the Digital Audio Access Protocol (DAAP[DAA 05]) and the Digital Media Access Protocol (DMAP).

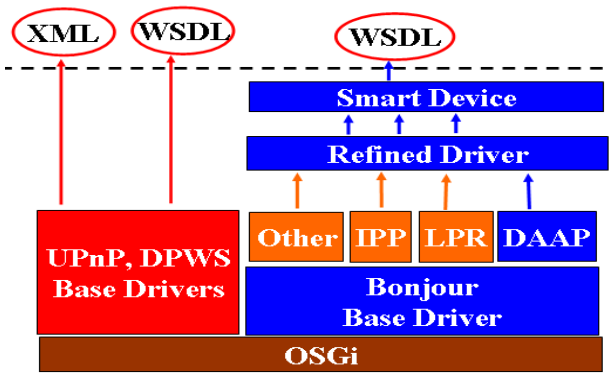


Figure 10.2: The Bonjour Base Driver compared with other Plug and Play Base Drivers

Thus, unlike UPnP, DPWS and IGRS which uses SOAP as a common invocation protocol, a Bonjour device have a diversity of protocols on top of multicast DNS. This multitude of interaction protocols prevents to have a unified layer to invoke services on a Bonjour device.

[1]http://files.multicastdns.org/draft-cheshire-dnsext-multicastdns.txt

In the HomeSOA [Bottaro 08a] framework, as shown in Figure 10.2, a Bonjour base driver is implemented, then a DAAP/DMAP layers are added to represent the *i*Tunes device profile. Furthermore, to interact with a Bonjour printer device for example, an IPP and LPR layers must be added on top of the Bonjour base driver as shown in Figure 10.2. Thus, since Bonjour only supports discovery and annunciation, an interaction layer must be added on top of the Bonjour base driver for each new device type supporting a different interaction protocol.

Thus, a possible realization to generate Bonjour ontologies can rely on the smart devices which expose their description in a WSDL. Therefore, the Bonjour OWL Writer will generate ontologies based on the WSDL-based description provided by the smart devices.

Once the ontologies are automatically generated, they are used by the ATOPAI framework to perform the matching between two equivalent device types.

10.2 ATOPAI

In this section, we detail the **A**lignment and annotation **T**ool framew**O**rk for **P**lug and pl**A**y **I**nteroperability (ATOPAI) which allows to perform the device matching between two generated ontologies by the OWL Writers.

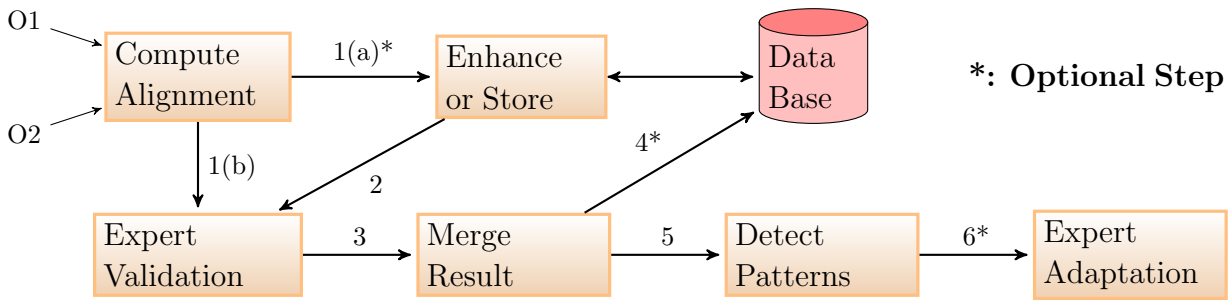


Figure 10.3: ATOPAI Supported Features

We implemented the ATOPAI framework on top of the Align API [INRIA] which already supports several basic matching techniques such as the *Ngram*, *Jaro*, *Hamming*, *Levenstein* and *SMOA*. We added the SMOA++ matching technique along with the structural enhancer to propagate similarity values between the entities of the alignment. We also added to the ATOPAI framework, a Graphical User Interface [Sabran 10] based on the Swing API to simplify the device matching to the expert which does not have to be an ontology expert to use our framework. A snapshot of the ATOPAI's GUI [Sabran 10] is showed in Figure 9.22.

The ATOPAI's GUI [Sabran 10] follows the MVC architecture (Model-View-Controller). The View-Controller part is mainly handled by the Java Swing toolkit. The displayed information is maintained up-to-date using an observer design pattern. The Model part is separated into several parts. First, we have a list of classes which represent each element from the ontologies device, service, action, variable. Second, the *RelationComponent* class represents a relation between two entities contained by the current alignment. This class provides simple methods for the GUI to display the tuples or to modify them. The *OntologyLoader* and the *AlignmentLoader* classes rely on the visitor design pattern, they allow to instantiate a data structure containing the information from the input ontologies or alignment.

The main supported features of the ATOPAI framework are shown in Figure 10.3. The expert loads two ontologies and triggers the automatic alignment matching. The result expressed in tuples (leftEntity, rightEntity, similarity value) can be stored into a MySQL (5.1) database, 1(a) in Figure 10.3. The actual alignment result

can be enhanced by querying the database. To enhance, the ATOPAI queries the stored tuples, if a tuple has a higher similarity value, then the actual computed similarity result, then the stored value is replaced with the actual similarity.

The alignment validation requires from the expert to add, remove or edit lines between the entities. ATOPAI allows to select two entities and then by clicking on the Add alignment button, a new tuple is added to the alignment result (leftEntity, rightEntity, 1). The new added alignment between two entities by the expert are considered valid. Therefore, we assign a similarity value of 1 to avoid removing the validated alignment when trimming, i.e. keeping the alignment having a similarity value high then a threshold t.

The Align API provides an "*OWLAxiomsRendererVisitor*" which transforms the alignment results expressed in the alignment rdf format, as shown in Listing B.3 in the appendix B.5, to an OWL alignment ontology containing equivalent relations instead of the similarity values. Such transformation allows ATOPAI to merge the three ontologies left, right and the alignment ontology into a single ontology containing the entities from both ontologies and the equivalent mappings between. The rules can then be applied on the single merged ontology to detect patterns. The rules are applied automatically and guide the expert in the final adaptation step.

We detail next the implementation of the main steps: ontology alignment, expert validation, patterns and adaptation.

10.2.1 Ontology Alignment

The Align API supports string-based matching techniques, however, such techniques clearly cannot detect synonyms and antonyms. Thus, we propose the SMOA++ [Mora 10] technique based on the external dictionary WordNet [Fellbaum 98]. We used the Rita[2] WordNet Java API, which allows to retrieve the synonyms, antonyms needed in our context for the ontology alignment.

ATOPAI supports the basic matching techniques along with SMOA++, however, it does not yet supports combining strategies by assigning confidence or weight values for each basic matching technique. We describe next, the implementation of SMOA++.

10.2.1.1 SMOA++

The SMOA++ technique relies on the substring search in the wordnet dictionary. It is inspired from the SMOA technique which searches for the biggest common substring between two strings to match. In SMOA++, first we apply the tokenization, to search for the biggest substring in WordNet and then we remove it from the string and search for another substring until none can be found.

```
1 public List<String> getSubStringsListFromWordNet(String str1){
2  int subString_index[] = new int[str1.length()]; //The start index of valid substrings
3   if(!wordnet.exists(str1)){
4     if(str1.length() > 3) { // Avoid propositions
5       for(int start = 0; start < str1.length()-2; start++){
6        for(int i = start + 2; i < str1.length(); i++){
7          if(wordnet.exists(str1.subString_index(start, i+1))){
8           if(subString_index[i] < (i - start)) subString_index[i] = (i - start);} } } } }
```

Listing 10.3: SMOA++ search for substrings in WordNet

Listing 10.3 shows a part of the implementation and the substring search in WordNet. We first test in [line 3] if the string exists in WordNet. If it exists, then, then there is no need to apply the substring trimming, we

[2] www.rednoise.org/rita/wordnet/documentation/index.htm

directly search for the synonyms and antonyms between the two strings to match. We search in [line 4] only for substrings with a length above than three characters to avoid searching for the propositions, such as to or in. Then, search for the substrings which exist in WordNet and keep the biggest common substring [line 8]. For example, between the *Create* and *eat*, the kept substring is *Create*. The start index of each found substring in kept in the subString_index table. The search continues until no substring is identified in WordNet. Then, based on the saved indexes, we extract the substrings and add them to the list of valid WordNet substrings.

The previous tokenization step returns two lists of substrings found in WordNet. The matching step consists in finding synonyms or antonyms in order to decide if the two string are equivalent or not. Using the Rita API, we can retrieve the synonyms and antonyms of each substring. Additionally, the Rita API returns the best position of a string in WordNet and whether it is used as a verb, noun, adjective or an adverb. We exploit the four possibilities when searching for synonyms. If a synonym is found between two substrings then the similarity value is set to 1 between the two. The Rita API also allows to retrieve if two strings are related with a coordinate relation, i.e. they share the same hypernym or superconcept. For example, "fork" and "spoon" are coordinate terms, they are both eating utensils. We also retrieve the list of coordinates and handle it as the synonyms list.

During the antonyms detection, we apply a first round of search between the two substrings lists. Using the Rita API, we check if there is at least one pair of antonym between two substrings from each list. If the first round did not detect any antonyms, we apply a second round, where we query the WordNet dictionary for the synonyms of each substring in the two lists. Then, for each synonyms, we retrieve its antonyms. If a pair of antonyms matched then the global similarity value is set to 0. Listing 10.4 shows a part of the implementation to retrieve the second round antonyms using the Rita WordNet API. the position *pos* and the integer both refer to the best position whether it is used as a verb, noun, adjective or an adverb.

```
1 public Set<String> getAntonyms2Round(String str1, int i, String pos) {
2  Set<String> synonyms = getSynonyms(str1, i);
3   for(String syn : synonyms){
4     antonyms =wordnet.getAllAntonyms(syn,pos);
5       if(antonyms != null) listOfAntonyms.add(antonyms); }
6  return listOfAntonyms;}
```

Listing 10.4: Second Round Antonyms Search

To detect the equality between two substrings, we rely on the *equalsIgnoreCase* method provided by the Java String API, however, other basic matching techniques can also be used instead.

10.2.1.2 Filters

As mentioned in section 9.4.1, the basic matching between an ontology O_1 having n elements and an ontology O_2 having m elements results in a matrix[n][m] holding similarity values between the entities. We filter these results by applying two types of filters, as detailed in section 9.4.1.2. The first keeps the best alignment similarity value in each line of the matrix. The implementation is simply applying a max function on each line of the matrix. The second filter applies a test on the resultant tuple (leftEntity, rightEntity, similarity) to check if the leftEntity and the rightEntity belongs to the same type, device-device, service-service, etc. If it the case then the similarity value is kept, else it is set to zero. A tuple holding a similarity value 0 will be removed from the alignment file when a trimming threshold is applied at the final result.

10.2.1.3 Similarity Propagation

In order to enhance similarity values between entities, we apply a similarity propagation based on the structure. After applying the filters, we revisit the alignment and enhance the similarity values by applying the Equation 9.4. We apply a down-top propagation, we enhance the similarity of the actions if the similarity values of the parameters are higher than a defined threshold set by the expert using ATOPAI. The propagation proceeds by enhancing the similarities of the services based on the actions. And finally, the similarity of the device which is based on the similarity of the services.

10.2.1.4 The WSDL Structure

As mentioned in section 10.1.1, the WSDL is flattened, therefore the actions parameters in the DPWS, IGRS and Bonjour ontologies will hold a path in their names as in *CreatePrintJobRequest/PrintTicket/Copies*. Therefore, when it is matched with another parameter from the UPnP ontology like the parameter *Copies*, we only consider for now the last element of the path (*Copies*).

Another strategy can also be used. For instance, we can keep the maximum found similarity between all the path elements and the UPnP matched parameter. For example, the matching between the two parameters will perform a SMOA++ matching on the following pairs of elements: (*CreatePrintJobRequest*, *Copies*), (*PrintTicket*, *Copies*), (*Copies*, *Copies*) and then keep only the maximum similarity. Applying the matching on each of the elements and not only the last element might allow to detect more mappings since the elements in the path are somehow semantically related with the last element of the path.

We detailed in section 9.4.1.4, the enhancer step, which consists in enhancing the similarity values between two services, for example, based on the similarity values of their actions and if there is a high similarity value between one of the services and the actions. The enhancer is not supported yet in the current version of ATOPAI.

We detail next the implementation of the pattern detection which annotates the ontology with composition relations between the actions.

10.2.2 Pattern Detection

The ATOPAI framework supports the pattern detection by triggering a set of pre-written rules specified in the Ontology Pre-Processing Language [Šváb Zamazal 10]. In order to detect the patterns detailed in section 9.4.3, we specified 12 rules as following: three rules are used to detect the Simple_Mapping with its three subcategories (Input, Output and Input_Output). Three rules are used to detect the Union_Mapping_One_to_N patterns which contains three subcategories (Input, Output and Input_Output). The sequential union pattern requires also three rules to detect such patterns along with three additional rules to detect the has_Next relation which specifies the sequential relation.

As mentioned in section 9.4.3, the N-to-M unions are almost impossible to adapt, therefore, we don't treat the pattern detection and adaptation of such unions. So far, ATOPAI only supports the One-to-N patterns detection and adaptation.

The pattern detection implementation takes a left ontology, a right ontology and an alignment ontology, then merges the three ontologies into a single one. The rules are then applied on the merged ontology to detect the patterns. This step also adds the new patterns properties such as *Simple_Mapping_Input* to the ontology which are used later by the the OPPL2 API[3] rules to connect the classes. The pattern detection implementation loads a file containing the 12 OPPL rules and then applies them on the ontology in a specific order since

[3] `http://oppl2.sourceforge.net`

the patterns depends on the previously detected patterns. As detailed in section 9.4.3, more precisely in the Listing 9.2, the union pattern detection is based on the *Simple_Mapping_Input* pattern detection. The *has_Next* property is also based on the union pattern as shown in Listing 9.3. The sequential union is based on both the union mapping and the *has_Next* property. Thus, the *Simple_Mapping* pattern is first detected, then the *Union_Mapping* pattern. The *has_Next* property is then found based on the *Union_Mapping*. And finally, the *Sequential_Union* patterns are detected based on the *has_Next* and the *Union_Mapping*.

Matching Concept Once the rules are applied, the ontology is automatically annotated with the new detected patterns properties. The next step consists in finding valid mappings and compositions based on the equivalent input/output parameters as detailed in section 9.4.3.2.

```
1 sequential_Mapping<OWLClass,OWLClass>=mergedOntology.getClassOfProperty("Sequential_1_to_n")
2 decideMatching(sequential_Mapping);
3 union_Mapping<OWLClass, OWLClass>= mergedOntology.getClassOfProperty("Union_1_to_n");
4 union_Mapping.removeCommonActions(sequential_Mapping);
5 decideMatching(union_Mapping);
```

Listing 10.5: Part of the Matching Concept Implementation

Even though, OPPL fits well into our needs to detect patterns, however it still lacks some expressiveness. For instance, OPPL does not support a cardinality feature, for example count the number of input parameters an action has or count the number of equivalent parameters two actions share. There is also no support for a *Not* clause on classes to retrieve for example a UPnP_Action class not having any output variable. A *Set* statement is also needed instead of only adding or removing an information. Loops and condition controls might also be useful. Thus, since OPPL does not support such features, we rely on the OWL API and Java code to detect the matching concepts along with the validity and the matching category between actions.

Listing 10.5 shows a part of the Java implementation of the matching concept. First, in [line 1], we retrieve all the classes related with the *Sequential_1_to_n*. The method *getClassOfProperty* returns a map containing pairs of OWL Class actions. The left element always refers to the UPnP ontology and the right ontology refers to DPWS, IGRS or Bonjour. So far, we only support DPWS.

The returned map is then used as an entry of the method *decideMatching* which applies the definition 9 on each sequential union 1 to n pair. The *decideMatching* method counts the number of input and output parameter each action has and the number of equivalent parameters. Then based on the decision Table 4.5, the method returns to the expert the decision of the matching.

We apply the same steps on the rest of the patterns. However, since the patterns are dependent on previously detected patterns, for example a *Sequential_1_to_n* is also a *Union_1_to_n*. Therefore, when applying *getClassOfProperty("Union_1_to_n")* on the ontology, it will also return the actions already treated in the previous step, in [line 2]. Thus, we delete from the returned map the list of previously sequential unions and then we apply the *deviceMatching* on the union mappings.

The detected patterns and the added annotation to the ontology are used as a filter to retrieve the action composition instead of only relying of the *equivalentTo* relation between the classes of the ontology. The method *getClassOfProperty("equivalentTo")* will return all the equivalent entities in the ontology, devices, services, actions and parameters and then it is up to the developer to test the entities' types in order to carry on with the matching concepts. Using specific relations added by the patterns such as *getClassOfProperty("Sequential_1_to_n")* guarantees that the returned entities are actions which makes it easier to the developer.

The *decideMatching* returns the decision of the matching concepts and then it is up to the expert to check for the non valid compositions and mappings if an adaptation is possible. The adaptation can require either

a default value or an adaptation behavior. However, the non valid compositions of mappings are removed manually one by one by the expert which selects the entities in ATOAPI and removes the lines between.

We present next, the adaptation which can be carried out by the expert to add an adaptation behavior or a default value.

10.2.3 Expert Adaptation with ATOPAI

Three types of adaptation are supported by ATOPAI which can be carried out by an expert. The first type allows to add a default value to an input or output parameter. The second type consists in adding an adaptation behavior as shown in the use case #1, section 9.4.4, between the TV devices. And finally, the last type allows to invoke an external OSGi service on the framework to provide some adaptation. The use case #2 provided in section 9.4.4 relies on an external service to convert a *ps* file type to a *pdf* type. We detail next the three adaptation types.

10.2.3.1 Add Meta Data

The first type of adaptation is applied on the actions parameters. The DPWS printing ticket used to print a document requires a true or false value for the *LastDocument* parameter. The DPWS standard printer profile allows to applications to send multiple documents to print for the same printing session identified by the *JobId* value. Thus, when a DPWS application send several documents, it has to specify using the *LastDocument* set to true that no other documents will be sent and the printer can process the Job and print the documents. The UPnP Printer does not support such feature and accepts only one document per session.

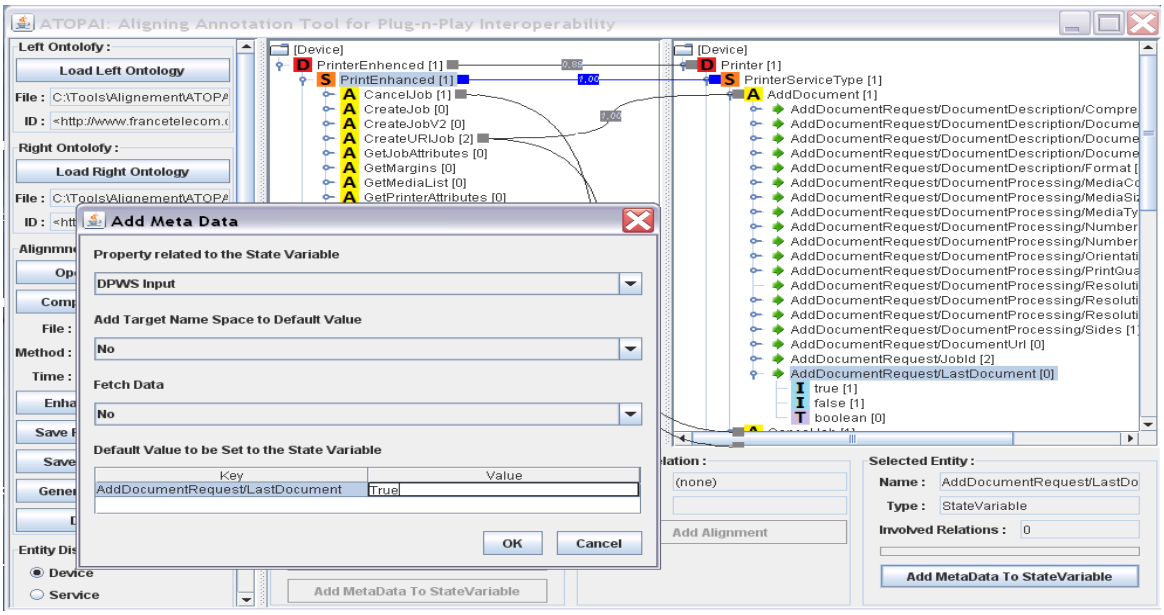


Figure 10.4: ATOPAI: Meta Data Adaptation

Therefore, since the UPnP applications will always print one document per session as supported by the UPnP standard printers specifications, the proxy can send the DPWS print command with the *LastDocument* DPWS parameter always set to *true*.

ATOPAI allows the expert to add a default value, through the meta data feature. As shown in Figure 10.4, the expert selects the parameter to adapt *LastDocument* and presses the *Add Meta Data To State Variable* (below-right), the meta data window pops up and the expert can set the value *true* as an input value of the

selected parameter. As soon as the expert validates, ATOPAI annotates the parameter in the ontology with the default value using the *has_Default_Value* owl data property to relate the parameter *LastDocument* with its default value *true*.

ATOPAI also supports adding a target name space to the parameter structure which is the case of the *GetPrinterElements* action, see Table 9.4. We set the input parameter to *tns:PrinterElements* to validate the mapping with the *UPnP.GetPrinterAttributes* action.

ATOAPI also supports fetching data from a URL and transforms it into a file to be used by another action. This feature is applied on the sequential mapping between the *CreateURIJob* and the *CreatePrintJob*, *SendDocument*. The *SendDocument* expects a file while the *CreateURIJob* contains the url location pointing at the file. Thus, the fetch feature allows to add an adaptation between the two actions, from url to document by retrieving the document from the url at runtime. Since, the fetch from url feature is a generic method and can be used for other cases, we decided to implement it in ATOPAI. However, it can also be used as an external service call instead. There is another possible sequential mapping between the printers, the *CreateURIJob* is equivalent to *CreatePrintJob* and *AddDocument* which takes a url. Thus, in this case the fetch from url is not used.

The meta data annotation allows to apply basic adaptation on the actions input and output parameters. However, not all the adaption can be resolved by setting default values. The adaptation can also require adding a behavior using loops or treatment on the values using high level operations on the ontology entities as shown next.

10.2.3.2 Adaptation API

The UPnP and DPWS API require an advanced knowledge in the plug and play technology in oder to apply an adaptation behavior and translation between the actions and their parameters. Applying a low-level adaptation by manipulating the Plug and Play protocols API can be a tedious and an error prone task. Thus, we specified the high level adaptation API to offer the expert the ability to specify a behavior relating two or more actions along with their related parameters independently from the UPnP and DPWS API. The expert manipulates the entities (actions or parameters) requiring adaptation through high level Java syntax-based operations such as set, get and invoke.

```
1 public interface Adaptation{
2  // Returns an action from the right ontology
3  public Action getRightAction(String actionName);
4  // Retrieves the value of the Input paramater parameterName
5  public Object getInputValue(String parameterName);
6  // Retrieves the value of the output parameter parameterName
7  public Object getRetValue(String parameterName);
8  // Sets the return value of an Action.
9  public void setRetValue(String ParameterName, Object ParameterValue);
10 // Sets the input value of an Action.
11 public void setInputValue(String ParameterName, Object ParameterValue);
12 public void invoke();} // Invokes the action
```

Listing 10.6: The Adaptation API Interface

The adaptation using ATOPAI is accomplished as follows. First, the expert selects the action to adapt on the left ontology. The adaptation behavior is executed by the proxy when the selected action is invoked. Since we currently support only a One-to-N union composition, then the expert can add only one adaptation behavior per left action. Once the left action is selected, as shown in Figure 10.5, the *Add Code* button will pop up an adaptation code window which consists of two parts. The upper part is reserved to the imported

classes while the Behavior Code part will hold the adaptation code added by the expert. Figure 10.5 shows the adaptation code between the two TVs overviewed in section 9.4.4, the added adaptation code is shown in Listing 9.6. The added code is saved in a file and the selected action from the ontology is annotated with the following data property *has_Adaptation_Code* which relates the specific action to the adaptation code file location. Such information is exploited during the code generation, the added code will be injected in specific positions in the templates. So far, we only support an adaptation code written in Java based syntax.

The interface of the Adaptation API is shown in Listing 10.6. It offers methods to select an action from the right ontology. It also allows to invoke any selected action. The input/output values of a selected action can be set or retrieved. To apply the methods on the selected action of the left ontology, the keyword *this* is used, see Listing 9.6.

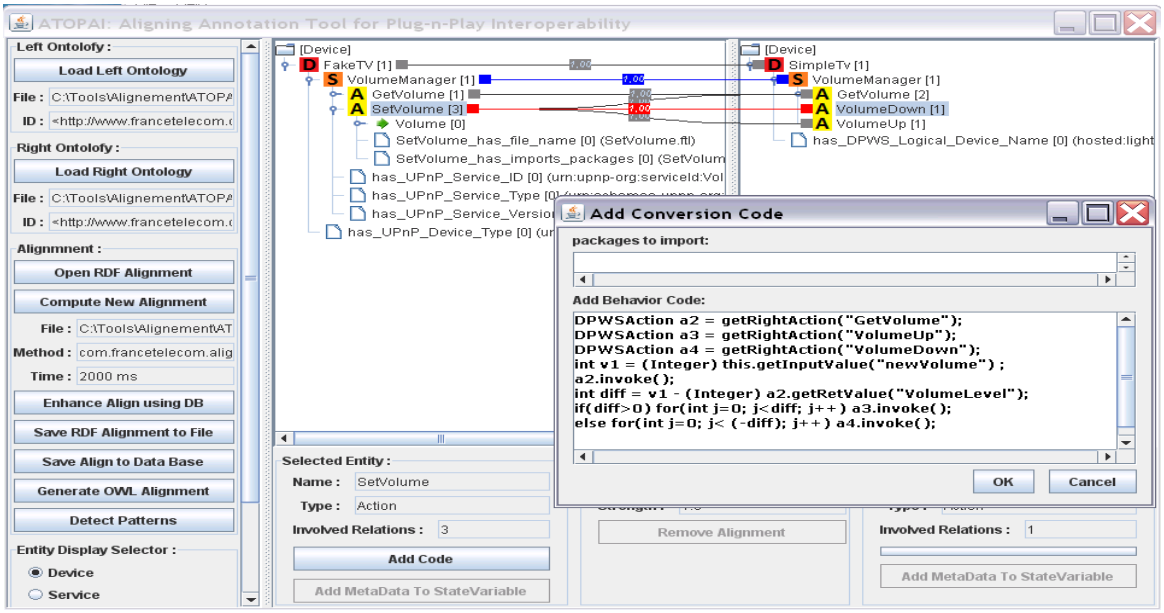


Figure 10.5: ATOPAI: Adaptation Code

The Adaptation API offers a high level entity manipulation to adapt action invocations. However, some adaptation might require an external conversion or transformation from a generic service such as the *ps* to *pdf* conversion. We detail next, the external service API which allows to invoke external services.

10.2.3.3 External Service API

The external service API is used by the expert to invoke an external service such as an OSGi service present on the framework. The use case #2 detailed in section 9.4.4 shows how the adaptation can be performed by invoking a general purpose service which converts a *ps* document into a *pdf*.

To use an external service, the expert relies on the API when defining its adaptation code exactly as in the adaptation API. The action in the ontology is also annotated with the reference to the code file. The added code is also injected in the templates during the code generation. The implementation of the External Service API, shown in Listing 10.7, relies on the reflection API (*java.lang.reflect*[4]) to search for the method to invoke along with its input/output parameters and values. moreover, the proxy subscribes and searches for the required service. Thus, the expert must first specify the service name or the class to search for on the OSGi framework. Then, the method to invoke along with the input parameters names and values (since the same method can

[4] `http://download.oracle.com/javase/6/docs/api/java/lang/reflect/package-summary.html`

have the same name but different signature). The returned values can also be retrieved. Listing 9.7 shows an adaptation code example using the External Service API.

The next section details how the ontology alignment is exploited to generate executable code which corresponds to a proxy providing interoperability between the two profiles in the ontology alignment.

```
1 public interface ExtService{
2  // Sets the class or service name to search for
3  public void setClassName(String className);
4  //Sets the method name to invoke on the className
5  public void setMethodToInvoke(String methodToInvoke);
6  //Sets the input value of the method to invoke
7  public void setInputValue(String parameterType, Object parameter);
8  // Retrieves the return object after the invocation
9  public Object getRetValue();
10 public void invoke();//Invokes the current set method on the class
11 public String getClassName();//Returns the Class name
12 public void setCurrentMethod(String currentMethodName);// Sets the current method to invoke
13 public String getCurrentMethod();}// Gets the current methodName to invoke.
```

Listing 10.7: External Service API

10.3 DOXEN

In this section, we provide details on the implementation of DOXEN and explain the proxy generation mechanism. We also give an insight on the management services supported by DOXEN and its generated proxies. Such services allow to carry out some basic management operations such as updating its configuration. Then, we describe through a use case in the experimentations section, how such management services can be used to carry out diagnostic operations. Such operations allow to detect basic malfunctions of applications interacting with the generated proxies in the digital home.

The DOXEN module takes an ontology alignment as an input, walks through the ontology and fills Java templates to generate Java code. The generated code is then compiled on the fly and packaged in an OSGi bundle (see section 3.3). Once started the OSGi bundle searches for the non-UPnP device type contained in the ontology alignment. When the non-UPnP device is found, the proxy first verifies that is supports the same services versions. Then, the proxy announces itself as a UPnP device on the network. When an application invokes the UPnP actions on the proxy, it will adapt the invocations and translates them to the equivalent non-UPnP device on the network.

DOXEN is installed in a home network, on a set-top-box for example. It listens to the non-UPnP device appearance on the network and based on its configuration containing the supported equivalent devices, the code generation can be triggered. Listing 10.8 shows a part of the local configuration file. It contains several cells, each cell contains a cell identifier and a UPnP device type and services along with their equivalent non-UPnP devices and services. Thus, when a DPWS printer appears on the network, DOXEN verifies that it announces the device type *PrinterEnanced* and supports the DPWS service *PrintEnhanced*. Then, DOXEN retrieves the correspondent ontology alignment *UPnP-DPWS-Printers.owl*, (see Listing 10.8) from the local repository and triggers the proxy generation. The same device and service names are only supported by the standard DPWS printer profile. Thus, each standard printer declaring itself with such names follows the standard DPWS printer operation names and parameters.

The proxy generation consists of three major steps. First, DOXEN visits the ontology and extracts the needed information to fill the templates with. The second step proceeds with the code generation and the final

step consists in compiling and packaging the generated code to build an executable OSGi bundle. We detail in section 10.3.1 the ontology visiting and the information extraction. In section 10.3.2, we explain the code generation and the template filling, then we provide in section 10.3.3 the OSGi bundle packaging.

```
1 <Cell>
2   <CID>3</CID>
3    <UPnPDevice>
4      <DeviceType>PrinterEnhanced</DeviceType>
5       <Services>
6         <Service>
7           <Type>PrintEnhanced</Type>
8           <Version>1</Version>
9         </Service>
10       </Services>
11    </UPnPDevice>
12    <DPWSDevice>
13     <DeviceType>PrintDeviceType</DeviceType>
14      <Services>
15        <Service>
16          <Type>PrinterServiceType</Type>
17          <Version>1.0</Version>
18        </Service>
19      </Services>
20    </DPWSDevice>
21  <OntologyIRI>/localRepo/UPnP-DPWS-Printers.owl</OntologyIRI>
22 </Cell>
```

Listing 10.8: Part of the DOXEN Configuration file

10.3.1 Ontology Visiting

DOXEN visits the ontology to extract necessary information for the proxy generation such as the name of the devices, services, actions and parameters. It also retrieves the properties between the entities such as equivalent parameters, union and sequential union compositions between actions. Additionally, the annotated meta data and behavior code added by the expert is taken into account.

DOXEN loads the alignment file and points to the entry point, the "UPnP_Device" OWL class. Each generated entity (by OWL Writers) in the ontology is a subclass of a well defined concept from the meta-model. In the lights ontology for example, (see Figure 8.3), *BinaryLight* (resp. *SimpleLight*) is a subclass of *UPnP_Device* (resp. *DPWS_Device*). The same applies on the rest of the entities in the ontology.

Using an OWL *Entity Visitor* which implements an OWLClassExpressionVisitor, OWLDataRangeVisitor and OWLIndividualVisitor (see OWL API 3.0 [OWL]), DOXEN explores the ontology then instantiates Java classes. Entities in the ontology are linked through object and data properties such as *equivalentTo*, *Union_1_to_n* or *has_Adaptation_Code*, therefore when the visitor explores an entity, it instantiate a Java Class depending on its OWL Super Class. For example, when the visitor encounters the *BinaryLight* entity, since it is a subclass of the *UPnP_Device* entity, then the visitor instantiates a UPnP Device class. Then, DOXEN continues visiting the ontology by relying on the properties between the entities. For example, the *BinaryLight* entity is related to the *SimpleLight* with the *equivalentTo* property. The *BinaryLight* entity is also related to the *SwitchPower* service entity with the *has_UPnP_Service*. Thus, based on such information, DOXEN visits the entities and instantiates the required Java classes needed for the code generation. Figure 10.6 shows the Java structure instantiated by the visitor.

All the classes in Figure 10.6 extends the abstract class *Entity*. Each class contains fields specific to the

entity, for example, the *UPnP_Service* contains the service id, type and version extracted from the ontology. The *UPnP_Action* also contains the action names and the list of their parameters.

The *addProperty* method allows to extract the properties between entities such as the *equivalentTo*, *Union_1_to_n* or *has_Adaptation_Code* and according to the property an operation is executed. For example, when the *has_Adaptation_Code* is encountered, DOXEN prepares to inject the code behavior in the template code, along with the Adaptation API dependencies. Moreover, the UPnP_Action class contains the two methods *orderList* and *findAction*. These two methods are used to order a sequential union composition. It applies the Algorithm 1 described in section 9.4.3. Additionally, the *generate_Impl()* allows to trigger the code generation detailed next.

10.3.2 Code Generation

Once the necessary information is extracted from the ontology, the code generation can be triggered. This step fills predefined Java templates with the information carried by the instantiated objects. We used the *FreeMarker* [5] template engine to generate the proxy Java code. For clarity of presentation, we refer to the instantiated classes holding information from the ontology as *objects* and by *Java files*, the generated files obtained by filling (.tpl) templates.

```
1 Template activatorTpl = cfg.getFreeMarkerConfig().getTemplate("templates/Activator.ftl");
2 Outs = new FileOutputStream(cfg.NewJavaFile("Activator"));
3 out = new OutputStreamWriter(Outs);
4 Map activatorInformationMap= new HashMap();
5 activatorInformationMap.put("package", cfg.GetClasspath());
6 activatorInformationMap.put("upnp_device_name", UPnP_Device.device_type);
7 activatorInformationMap.put("dpws_device_name", DPWS_Device.logical_device_name);
8 activatorTpl.process(activatorInformationMap, out);
```

Listing 10.9: Preparing the Activator Information for Template Filling

Once the ontology is visited and the classes are instantiated, DOXEN starts preparing the templates to be filled. Listing 10.9 contains a part of the implementation. The first line in the Listing 10.9 loads the Activator template to be filled. The lines 2,3 creates an empty Java File "Activator.java" on the disk in a specific location contained in the configurator (cfg). The *activatorInformationMap* in line 4 will hold the necessary information to be filled in the template such as the package name, and the upnp device name to publish and the dpws device type to search for. Lines 6 and 7 fill the map with the information from the instantiated Java structure, more specifically from the UPnP_Device and DPWS_Device classes fields, see Figure 10.6. And finally, the line 8 invokes the FreeMarker template engine in order to replace in the template the $*upnp_device_name* and $*dpws_device_name* with the values contained in the map as shown in Listing 10.10.

```
1 System.out.println("1-Activating Bundle: Proxy UPnP-DPWS ${upnp_device_name} Device");
2 System.out.println("2-Searching for an equivalent DPWS ${dpws_device_name} Device");...
3 if(foundDpwsType.equalsIgnoreCase("${dpws_device_name}")) {
4  System.out.println("!! 3- Proxy: DPWS ${dpws_device_name} device found !!");..}
5 //Invoking other generated classes
6 deviceProxy = new ${upnp_device_name}Device(Ctxt, dpwsDevice, deviceTNS);...
7 System.out.println("!! 6- Proxy: Publishing UPnP ${upnp_device_name} device !!");
8 serviceRegistration=Ctxt.registerService(UPnPDevice.class.getName(),deviceProxy, dict);
```

Listing 10.10: Part of the Activator Template

Figure 10.7 shows the Java generation dependency between classes. DOXEN starts by filling the *Activator* template which is the *main* class of an OSGi bundle, see section 3.3. The Activator.java file holds the compatible

[5] http://freemarker.sourceforge.net/

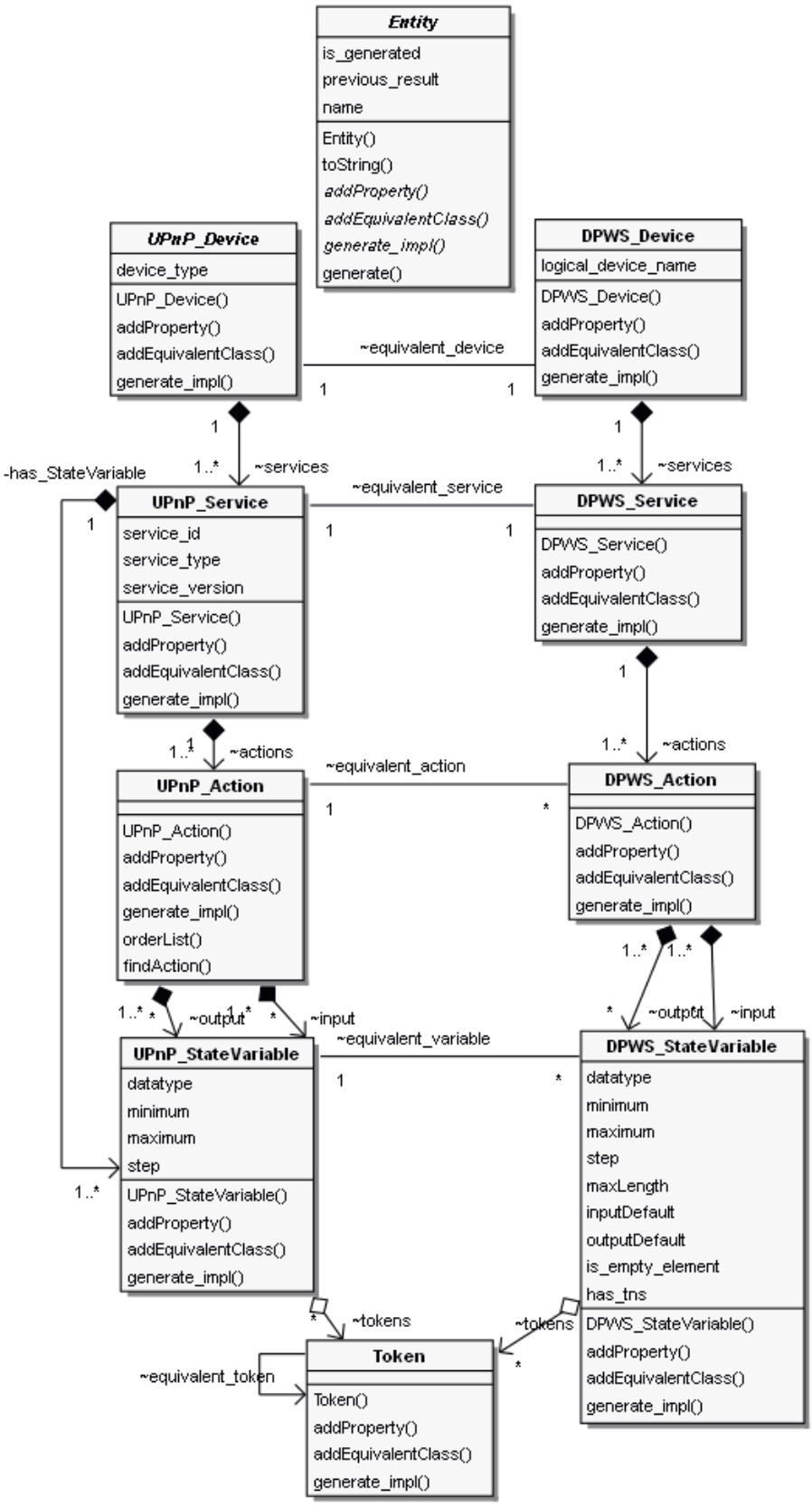


Figure 10.6: DOXEN Proxy Generation Architecture (Simplified)

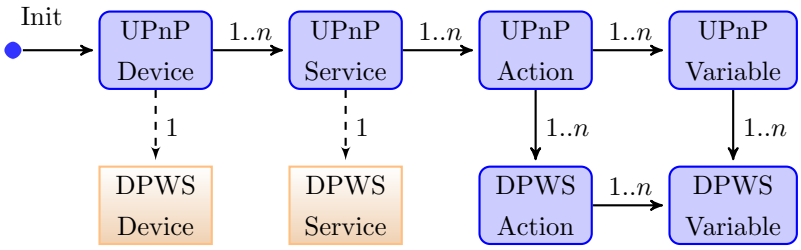


Figure 10.7: Code Generation Dependency

UPnP device type to publish ((UPnP) *BinaryLight*) and the DPWS device type to search for and bind ((DPWS) *SimpleLight*). Therefore, there is no need to generate Java files for the DPWS device and DPWS services.

However, the Activator.java refers to the UPnP_Device.java, thus the generation of the UPnP_Device.java is triggered by calling the *generate_impl()* method of the UPnP device object of the structure in Figure 10.6. The *generate_impl()* method of the UPnP_Device will first prepare the device template file then loads in a map some information related to the device just like with activator template filling in the Listing 10.9. Once the information is loaded in the map, the UPnP_Device triggers for each of its UPnP_Services the *generate_impl()* method as shown in Listing 10.11 line 6. The services names are retrieved and then used in the service registration, since a UPnP device also registers its services in OSGi. The service generation fills the service templates and triggers the actions generation. The dependency between the classes is shown in Figure 10.7. The generation of the UPnP_Device depends on the UPnP_Service which depends on the UPnP_Action and UPnP_StateVariable. During the action generation, the sequential actions are ordered (see algorithm 1) in a list and passed to the templates. The UPnP_StateVariable triggers the generation of equivalent DPWS State Variable Java files. The DPWS_Action can also triggers the input/output DPWS State variable Java files. This occurs when default values are added to DPWS state variables having no equivalence with a UPnP State Variable, such as the *LastDocument* parameter.

The expert added code is actually saved in template files and then injected in the actions templates using the <#include "$code"> feature of the *FreeMarker* template engine. Finally, DOXEN generates additional files such as the *DPWSElementFactory* which implements methods used by the actions to handle the conversion between the flat UPnP parameters and the structural elements of the WSDL.

```
1 deviceMap.put("package", cfg.GetClasspath());
2 deviceMap.put("upnp_device_name", name);
3 //Generate the device's services and retrieve their names to include them in the template
4 ArrayList<HashMap> services_namesMap = new ArrayList<HashMap>(services.size());
5  for (UPnP_Service service : services) {
6   service_names = (HashMap) service.generate(cfg);
7   if (service_names != null) services_namesMap.add(service_names);}...
```

Listing 10.11: Part of the UPnP Proxy Device Generation

Now that the Java files are generated, the final step, detailed next, consists in compiling the code and package it into an OSGi bundle.

10.3.3 Compiling and Bundle Generation

Once all the code source is generated, DOXEN compiles it on the fly using the Janino[6] compiler. We chose to use *Janino* since it is a light compiler (569 KB) compared to other compilers such as the Sun JDK compiler.

[6] www.janino.net

The *Janino* compiler is used mainly to compile and execute the compiled code on the fly. Therefore, we added an extra method *generateBytecodes_extended* to return the byte code of each compiled file.

We rely on two class loaders to compile as shown in Listing 10.12 [lines 1-2], the DOXEN Class loader and an additional *urlsOfJar* array which contains urls pointing to several Jar packages needed to compile. The line 3 starts the compilation by passing the *Activator.java* file, then *Janino* loads and compiles the dependent classes in the other generated files. Once the source is compiled, we write the byteCode in *.class* files and then we package all the classes in a Jar.

```
1 urlClassLoader = new URLClassLoader(urlsOfJar, doxenClassLoader);
2 JavaSourceClassLoader cl = new JavaSourceClassLoader(urlClassLoader, srcDirs, encoding,
      DebuggingInformation.DEFAULT_DEBUGGING_INFORMATION);
3   Map_ByteCode = (HashMap) cl.generateBytecodes_extended(cfg.GetBundleActivator());
4   Iterator it_byteCode = Map_ByteCode.entrySet().iterator();
5   int i =0;
6   File[] ToBeJared = new File[Map_ByteCode.size()] ;
7   while (it_byteCode.hasNext()) {
8     Map.Entry pairs = (Map.Entry)it_byteCode.next();
9     String Class_File_name = pairs.getKey().toString().replace('.', '/');
10    File F = new File(cfg.Target_Dir+Class_File_name+".class");
11    ToBeJared[i] = new File(cfg.Target_Dir+Class_File_name+".class");
12    OutputStream out2 = new FileOutputStream(F);
13    out2.write((byte[])pairs.getValue());
14    out2.close();
15    i++;}
16 CreateJarFile CJF = new CreateJarFile();
17 CJF.createJarArchive(new File(cfg.Generated_Bundle_Target), ToBeJared, cfg);
```

Listing 10.12: Compile Generated Code

Once the classes are packaged in a Jar file, we write the OSGi manifest which contains several kind of information. The general information exposes the bundle name, version and manufacturer. Most importantly, the manifest contains the name of the activator class along with the imported packages needed for the execution of the bundle, and the exported packages which can be shared with other bundles, for example, the UPnP proxy packages can be shared by the other bundles.

DOXEN can finally install the generated proxy on the OSGi framework and start the proxy, see section 3.3. When started, the proxy announce its description on the network and searches for a specific DPWS device to interact with.

So far, we only support Java/OSGi proxies providing interoperability between the UPnP and the DPWS profiles. In order to support additional protocols, another set of templates must be developed. As shown in the previous section, the templates contain code to call operations of the OSGi framework and the Plug and Play Base Drivers. Thus, their would be three types of templates to generate three types of proxies (UPnP-DPWS, UPnP-IGRS, UPnP-Bonjour) to resolve the heterogeneity between the four protocols. The advantage in using template ontology based code generation consists in writing once the templates and use anywhere with any alignment ontology.

The current version of DOXEN generates proxies and installs them on the same OSGi framework as DOXEN due to the code compiling class loader dependency and the bundle resolution. The compilation operation relies on the class loader of DOXEN and other Jars present on the framework. The bundle resolution also depends on other local bundles exporting their packages on the framework. However, a solution for a remote deployment can be based on the ROCS framework [Frénot 10].

In section 2.4.3, we gave an insight on the importance of the device and application management for an operator, a device manufacturer and an application vendor. Therefore, we also provide DOXEN and its generated proxies with management services in order to allow remote administration and action invocation. We detail next, DOXEN's supported services and capabilities. A use case in the experimentation section, shows how such management services can be used to carry out diagnostic operation. Such operations allow to detect basic malfunctions of applications interacting with the generated proxies in the digital home.

10.3.4 DOXEN's Supported Capabilities

DOXEN is deployed in the digital home and automatically generates proxies upon the detection of non-UPnP devices equivalent to UPnP standard devices. However, as mentioned is section 2.4.3, the remote administration of devices and applications is essential for telecoms operators and device manufacturers. The remote administration offers a lot of advantages [Lupton 07, UPnP 11, Broadband] such as applying configuration update and maintenance operations. Moreover, administration operations can be used to carry out diagnostic operations to detect an improper network or connectivity configuration, or even malfunctioning ubiquitous applications and devices.

Additionally, DOXEN relies on its configuration containing equivalent devices to generate a proxy, see Listing 10.8. Such generation is based on the templates and the alignment ontologies. Furthermore, DOXEN is expected to evolve in time to support additional proxy types generation. For instance, the operator can validate a new device matching between two new proposed devices, thus, DOXEN is expected to update its local configuration to support additional device types. Moreover, a new protocol can arise, thus, new templates need to be added in order to support a transparent interaction with the new protocol's devices.

We detail next, the capabilities and services provided by DOXEN.

10.3.4.1 Two Proxy Generation Modes

DOXEN supports two proxy generation modes: *brutal* and *on demand*. The *brutal* mode consists in generating the adequate proxies on the appearance of non-UPnP devices on the network. The implementation of the *brutal* method relies on registering a service listener for the DPWS devices.

As for the *on demand* mode, DOXEN listens to the applications requests (search or subscribe) for a UPnP device using the service hooks feature detailed in section 3.3.1.3. Once a UPnP device request is intercepted, a UPnP Printer request for example. Based on the request information (device type, meta-data information), DOXEN queries its configuration to check its non-UPnP equivalent device types and versions. Then, DOXEN scans the network searching for the specific non-UPnP device, a DPWS equivalent printer for example. Once found, the proxy generation is carried out as described previously. The *on demand* mode can be used for example when DOXEN is deployed on low energy and scarce resource devices such as smart phones. DOXEN will trigger the proxy generation only upon applications requests.

10.3.4.2 Uniqueness In Interaction

Consider two DPWS lights are detected in the network, DOXEN will generate two UPnP-DPWS light proxies since each proxy reflects the same state of the real DPWS device. Therefore, the invocations received by the proxy must be sent to one specific real device. Therefore, each generated proxy must be handling a unique DPWS light device. Therefore, DOXEN must guarantee the uniqueness in interaction.

To achieve such uniqueness, each started proxy interacts with DOXEN to check if other UPnP-DPWS lights proxies are already generated. If it is the case, the newly generated proxy receives a list of the real devices unique identifiers already handled by actual UPnP-DPWS lights proxies. Based on these communicated identifiers, the

new generated proxy will search for a DPWS light having an identifier not included in the list. Once found, the proxy notifies DOXEN about the identifier which updates its list of identifies. When the DPWS Light leaves the network, the proxy handling the device sends a notification to DOXEN about the device departure, then the proxy stops. DOXEN re-updates its unique identifier list by removing the received one.

10.3.4.3 DOXEN Management Services

DOXEN announces its management services as UPnP Services on the network to share its capabilities with other management applications in the digital home. Additionally, exposing DOXEN as a UPnP service provides the operator with the ability to remotely administrate DOXEN for example through the CWMP protocol mentioned in section 2.4.3. The administration operations vary from updating templates, configuration and ontology alignment files to troubleshooting and diagnostics.

DOXEN supports the three following UPnP services:

- DMS: The DOXEN management service offers several management operations such as start, stop and update DOXEN, retrieve and set the proxy generation mode. Other operations also allow to add, remove or update an ontology alignment, a template code file or a list of supported equivalent devices and services. Furthermore, DOXEN can also be invoked to return a list of its generated proxies.

- BMS: The basic management service [UPnP 10a] as overviewed in section 2.4.3 is a standard UPnP service which allows to carry out basic management operations to test the network connectivity through ping and traceroute actions for example.

  The current version of DOXEN implements the BMS actions such as the ping and traceroute by calling an exec process based on a Windows XP OS. A future version of DOXEN is currently under development and uses the SIGAR[7] API which offers common methods to extract networking information and to perform operations like a ping and a traceroute on different operating systems.

- CMS: The configuration management service [UPnP 10b] is also a standard UPnP service, it returns several information such as the device type and manufacturer, the network configuration of the machine where it is deployed. For example, CMS reveals the number of active network interface along with their IP and MAC addresses, the DNS and DHCP hosts. CMS also exposes general information about the physical device and its manufacturer. Information about the operating system can also be provided.

  The current version of DOXEN relies on the exec process and the methods of the *java.net* package. DOXEN retrieves the following information from the underlying device and the network: the number of sent/received bytes using the ("netstat -e") exec process, the IP and MAC address, the subNetMask, the default gateway IP, the number of existing interfaces and also detects which interfaces are up and the type network's card type, WIFI or LAN.

Development Effort Table 10.1 shows the development effort of each module expressed in lines of code. The UPnP OWL Writer is achieved in 582 LoC while the DPWS OWL Writer has 1119 LoC. The difference is due to the fact that in the DPWS OWL Writer, the WSDL service description file need to be retrieved and parsed along with the XSD file. Moreover, as mentioned in section 10.1.1, the WSDL and the XSD allow to have complex and structured elements which makes the ontology generation even more complex.

The ATOPAI framework and the GUI with the four steps mentioned in section 9.4 are implemented with 6 727 LoC. The implementation of DOXEN is around 2000 LoC, however, when adding the three management services BMS, CMS and DMS, the implementation reaches 11 637 LoC. The templates used to generated

[7] `http://support.hyperic.com/display/SIGAR/Home`

the proxy contain 1233 LoC, when the BMS and CMS management services are supported on the proxy, the implementation reaches 6 884 LoC.

Module	Lines of Code
UPnP OWL Writer	582
DPWS OWL Writer	1 119
ATOPAI	6 727
DOXEN	2 385
DOXEN-DM	11 637
Proxy Templates	1 233
Proxy-DM Templates	6 884

Table 10.1: LoC of each module

The next section draws the experimentations we carried out to validate our approach. It also exposes in a use case, how the combination of the BMS and the CMS management services can be used to perform basic diagnostic operations.

10.4 Experimentations

We validated our approach on 6 existing devices which we took on the shelf, we also implemented fake devices in order to carry out additional tests. Figure 10.8 shows the used devices and the topology. We used each existing DPWS device having an equivalent UPnP Device. We found a DPWS Light device proposed by the SOA4D [SOA4D b] project, a DPWS Clock device proposed by the WS4D[8] and a DPWS Printer, the HP 4515x. We used their equivalent UPnP devices types the *intel* UPnP Light[9], the UPnP Felix Clock [Apache b] and the UPnP printer standard profile [UPnP 06a].

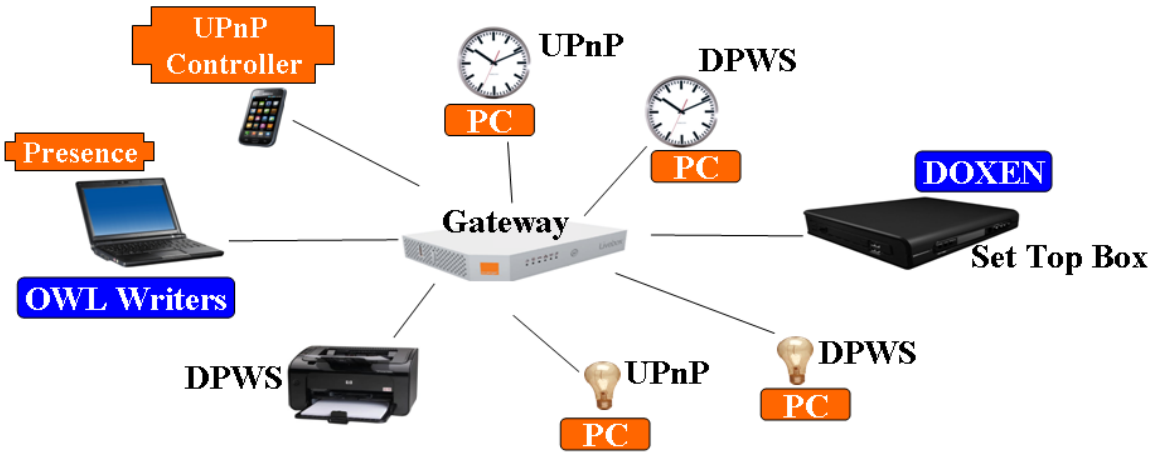


Figure 10.8: Experimentation Devices and Topology

The clocks and lights are OSGi devices while the printer is a real physical device. We deployed the OSGi devices on a PC on top of the Apache Felix OSGi implementation. We also used an Internet Gateway Device, the *Orange Livebox* to connect the devices in a star like topology. The DOXEN module is deployed on a

[8] www.ws4d.org
[9] http://software.intel.com/en-us/articles/intel-tools-for-upnp-technologies/

SodaVille (Set-Top-Box) with an Apache Felix OSGi Framework along with the UPnP Apache Felix [Apache b] and the SOA4D [SOA4D b] DPWS Base Drivers. The OWL Writers are deployed on a laptop. On the arrival of a UPnP or DPWS devices, the OWL Writers check if an ontology is already generated for this specific device type. The ontologies are automatically generated for each new device type. Using the ATOPAI framework, we successfully aligned the ontologies of these three device types, Lights, Clocks and Printers. The valid alignments are saved on the Set-Top-Box.

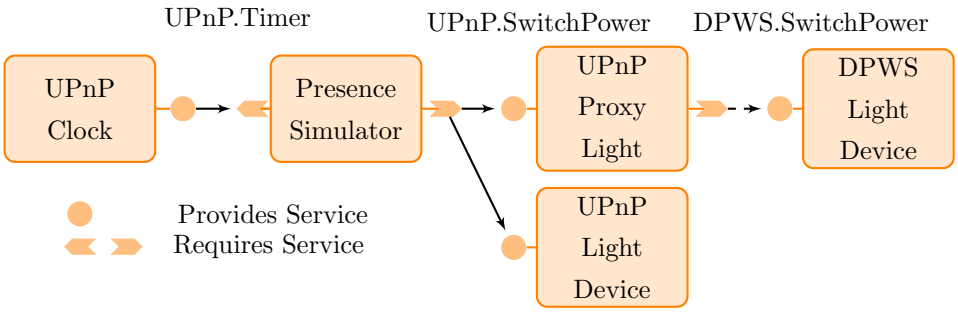


Figure 10.9: Presence Simulator application Scenario

We also implemented and deployed on the laptop, a UPnP Presence application, shown in Figure 10.9. The application detects a UPnP Clock and turns on/off all the lights at a given time of the day. The application is inspired from the scenario of chapter 2, the lights are turned off or on at a certain time or when there is no movement detected in the room.

We used the *intel* UPnP Felix Light and the SOA4D DPWS light. On the appearance of the DPWS Light, DOXEN triggers the proxy generation based on the valid ontology alignment. Once the UPnP-DPWS Proxy Light is installed and started, the presence application binds to the UPnP-DPWS proxy and controls the DPWS light as a UPnP light through the generated proxy. The adaptation process is transparent to the application which uses the same actions and parameters to interact with the DPWS Light.

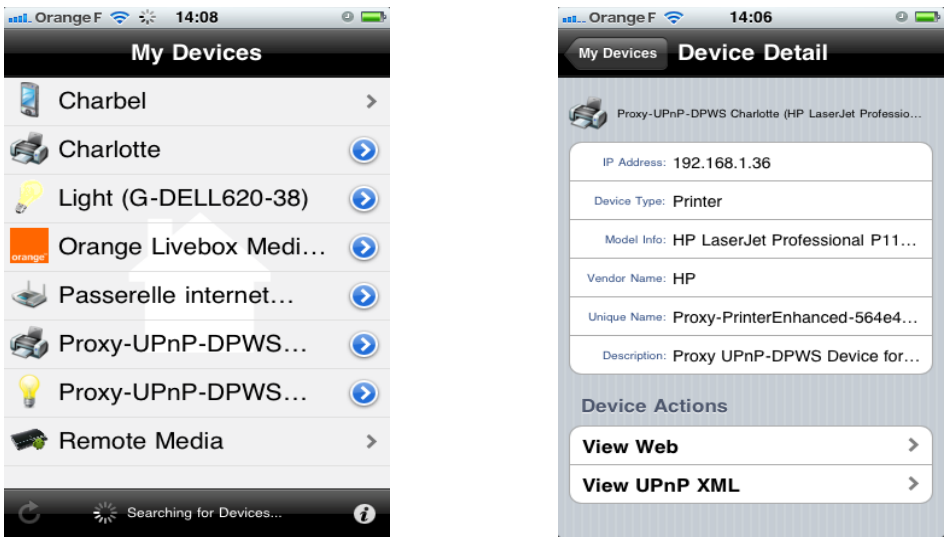


Figure 10.10: UPnP-DPWS Proxies detected by the MyDevices Application

All the generated UPnP proxies can be discovered as real devices using any UPnP Control Point such as the Felix UPnP Control Point which offers a generic interface to control any UPnP device. We are also able

to discover the UPnP proxies using the *My Devices*[10] application which discovers UPnP and Bonjour Devices as shown in Figure 10.10. The *My Devices* application is deployed on an 3GS iPhone. The left side figure shows a part of the available devices in our experimentation, while the right part figure shows a part of the information announced by the UPnP-DPWS Printer Proxy which exposes the same information retrieved from the real DPWS printer.

The *My Devices* control point allows to discover UPnP devices but does not support action invocation. Therefore, we implemented a "Home Controller" Application on a *Samsung GalaxyS Android* (2.2) smart phone which interacts **only** with **UPnP** standard Lights and Printers. Screen shots of the application are shown in Figure B.3 and Figure B.4 in the appendix B.6. On the *GalaxyS*, we only deployed a UPnP *Cybergarage*[11] stack. On the appearance of the DPWS light [SOA4D b], the DPWS WS4D clock or the DPWS HP 4515x Printer, DOXEN generates at runtime a UPnP-DPWS proxy for each device. The application controlled the DPWS Light and Printer devices through the generated UPnP proxies which transfers the invocations to the real DPWS devices. On the HP 4515x printer, we are successfully able to print a file(pdf, txt, ps), cancel a job, retrieve the job and the printer status.

We detail next a basic diagnostic use case showing how the BMS and CMS can be used to perform basic troubleshooting operations.

10.4.0.4 Basic Connectivity Diagnostic Use Case

The basic and configuration management services are also supported by each generated proxy. The UPnP standard CMS service [UPnP 10b] exposes only information about its hosted device. On the generated proxies, we extended the CMS to additionally expose networking configuration information of the real device such as the IP and MAC address, the DNS and DHCP servers along with the general device information (manufacturer, unique ID, etc). Thus, each generated proxy exposes the following services types:

- A set of specific services related to the device type and domain. Such specific services are generated based on the ontology alignment.
- A Basic Management Service [UPnP 10a] which provides basic operations such as the reboot, ping and traceroute.
- A Configuration Management Service [UPnP 10b] which provides two sets of information. The first refers to the hosted device while the second set provides information from the real DPWS device it is proxifying.

  To retrieve networking information from the real DPWS device, the proxy relies on the real device description containing general information such as the manufacturer, model etc. The proxy also retrieves the IP address of the DPWS device from the DPWS base driver. The MAC address is retrieved by applying two operations, first the proxy pings the DPWS device and then performs an ARP request. The proxy also detects the interface name of its underlying device used to communicate with the real device.

Such extension provides more visibility to UPnP monitoring applications such as the SLC4DH [Chazalet 11] and the DomVision [Petit 11] which monitor the UPnP devices in the digital home. It also gives the operator through the CWMP-UPnP Bridge [Broadband 10b, Lupton 07] additional information on the non-UPnP devices present in the digital home and handled by the generated proxies. Such extra information are exposed by the extended CMS service which provides information about the device hosting the proxy and the real device.

Adding the management services BMS and CMS to the generated proxy allows to diagnostic basic connectivity malfunctions as described in the following use case. The diagnostic process and algorithms are out of

[10] `http://itunes.apple.com/us/app/mydevices/id339268052?mt=8`
[11] `http://www.cybergarage.org`

scope of this work. The aim of this use case is to overview the benefits of exposing the generated proxies with a BMS and an extended CMS service providing information about the proxy's hosted device and the real device.

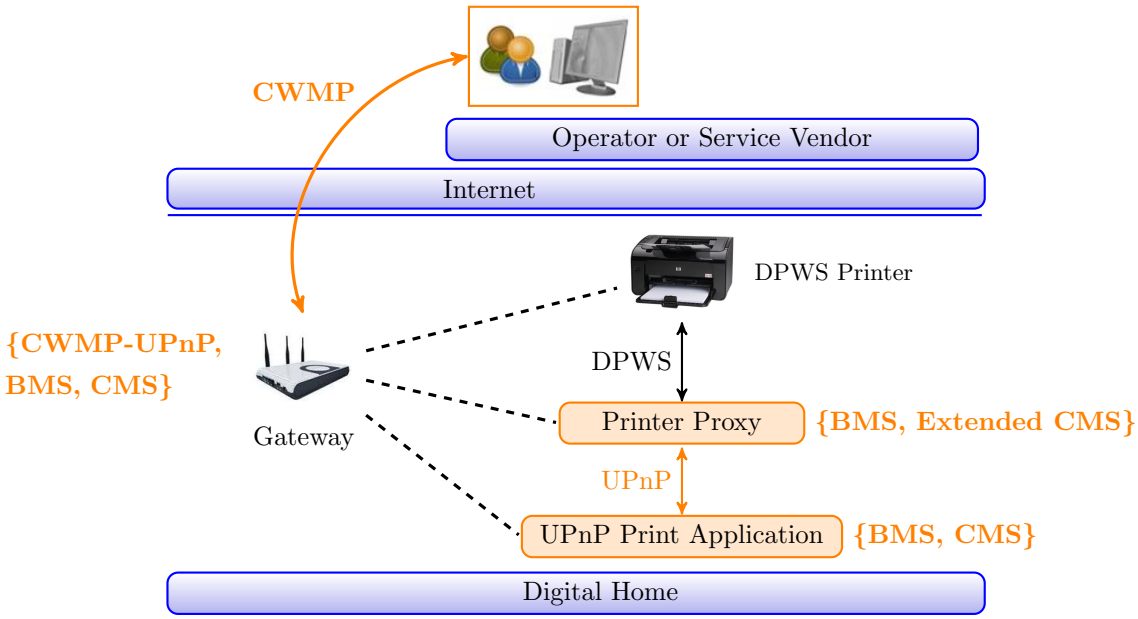


Figure 10.11: Basic Diagnostic Use Case

Figure 10.11 shows a DPWS printer device, a UPnP Internet Gateway device, a UPnP-DPWS printer proxy and a UPnP print application. The device hosting the printer proxy will be referred to as the "proxy-device" while the device hosting the application will be referred to as the "application-device". The application and the proxy can be hosted on several devices in the digital home such as a Set-Top-Box, a *GuruPlug*[12] or a PC. Figure 10.11 also shows the supported services by each device.

The Internet Gateway device is connected to all the devices in a star like topology. The Gateway supports the CWMP-UPnP [Lupton 07, Broadband 10b] Device Management Proxy which allows to transfer action invocation from the operator's site to the devices supporting UPnP Basic Management services. The gateway supports the BMS and CMS services. The printer proxy supports the BMS and the extended CMS exposing networking information about its hosted device and the DPWS printer. Finally, the printer application-device (not the UPnP application) supports the BMS and CMS services.

Now consider, the following malfunction, the end-user is unable to print, however he was able to print few hours ago. The end-user "Edouard", calls the operator's *Hotline* for an assistance. The *Hotline assistant* "Alice", connects to Edouard's gateway through the *CWMP* protocol. The CWMP-UPnP [Broadband 10b] Bridge allows Alice to forward action invocation to the UPnP Devices. For example, Alice can perform through the CWMP-UPnP Bridge a ping operation on the proxy BMS service or retrieve the information exposed by the extended CMS on the proxy. The extended CMS exposes the IP and MAC address of the hosted proxy device and the DPWS printer. This extension offers Alice an extra visibility and becomes capable to ping the printer and the proxy device through the CWMP-UPnP [Broadband 10b] bridge.

Alice remotely performs the following operations through the CWMP-UPnP Bridge and relies her diagnostic on their response:

1. Gateway.ping ("Application-Device"). The gateway pings the "Application-Device".

2. Gateway.ping ("Printer").

[12] www.globalscaletechnologies.com/t-guruplugdetails.aspx

3. Gateway.ping ("Proxy-Device").
4. Application-Device.ping ("Proxy-Device").
5. Proxy.ping ("Printer").

The operations #1, #2 and #3 are used to test if all devices are connected to the Gateway. The operation #4 tests if the UPnP application reaches the proxy. And the operation #5 tests if the Proxy can reach the Printer.

When Edouard reports the problem to Alice, she asks him specific questions such as "Does your UPnP application shows the UPnP Printer?" and then according to Edouard's answers, Alice performs diagnostic operations to detect the potential malfunctions:

1. Edouard: "*My UPnP Printer Application is not showing the UPnP Printer.*".

   For such error and based on the end-user feedback, there is the following potential malfunctions:

   - Connectivity Failure between the Application and the Proxy. It is verified if the operation #4 returned an error. Clearly, if there is no connectivity between the application and the proxy, the application will not receive the proxy annunciation and will be unable to interact with.
   - The DPWS Printer is not connected. Thus, the proxy did not detect the printer. It is verified if the operation #5 returns an error.

2. Edouard: "*I am able to select the UPnP Printer using my UPnP Printer Application. But it is still not printing*".

   - Printer was suddenly unplugged from power. As mentioned in section 9.3.3, the proxy's state depends on the real device's state. Thus, when the DPWS printer device announces its departure and leaves the network, the proxy receiving the DPWS announcement also announces it departure. Therefore, if the DPWS printer leaves the network, the proxy announces also its departure. However, if the printer was suddenly unplugged from the power switch, then it did not send the departure announcement. Since it is not received by the proxy, then the proxy remains present on the network and can detected by the UPnP Printer Application. To detect this problem, the operations #2 and #5 fails.
   - No Connectivity between the proxy device and the printer. Thus, the proxy cannot invoke the action on the printer, the physical link is down. The results of the operations #2 and #3 allow to detect whether the problem lies in the Gateway-Printer or in the Gateway-Proxy connectivity.

Other specific printer errors and specific DPWS devices messages are intercepted by the proxy and forwarded to the UPnP applications as UPnP Exceptions, for example, the "Tray Paper Empty", "Paper Jam" or "0% Ink".

We provided in this paragraph only a part of the experimentation regarding the interoperability, We show in [El Kaed 11c] how our approach can be mapped with other diagnostic and monitoring tools in the digital home.

We detail in the next chapter, the performance evaluation of our proposed modules, the OWL Writers, The Device Matching and DOXEN.

Chapter 11

Evaluations

"If that's an evaluation of what we're capable of doing, ... that's a good sign."

– Jon Gruden

Contents

11.1 OWL Writer **181**
11.2 Device Matching **186**
11.3 DOXEN **194**
11.4 Discussion **196**
11.5 Conclusion **197**

We evaluate in this chapter our proposed approach to resolve the plug and play devices heterogeneity. In section 11.1, we evaluate the ontology generation on a PC and a Set-Top-Box. We outline in section 11.2, a comparison between our proposed method SMOA++ and the other syntax based methods. Then, we present the ontology alignment results of equivalent device types based on the SMOA and the SMOA++ techniques. In section 11.3, the proxy generation evaluation is provided. And finally, we discuss in section 11.4 the evaluation of our approach with respect to the proposed approaches in the literature.

11.1 OWL Writer

We tested the OWL Writers on an Intel x86 Centrino Duo Core PC, with a 2 GHz clock frequency and 1 GB RAM capacity. We were also capable of deploying and testing the OWL Writers on a *SodaVille* Set-Top-Box with an intel Atom 1.2 GHz frequency and (256+128) MB of RAM. The evaluation on the PC provides the reader with an insight on the OWL Writers' performance, while the evaluation on the *SodaVille* aims to prove that our modules can be deployed on real embedded devices in the digital home.

The OWL Writers generated ontologies for the UPnP and DPWS Lights, Clocks and Printers. We used a UPnP standard light and printer profile proposed by the UPnP Forum. The UPnP Clock profile is proposed by Apache Felix as a fake device. The DPWS Light profile is found at the SOA4D [SOA4D b] forge, while the DPWS clock profile can be retrieved from the WS4D[1] forge. The DPWS standard printer is proposed by Microsoft. Thus, the six devices are on the "shelf devices" which can be purchased or are prototypes of development projects. In other terms, we used existing devices to validate our approach without modifying and arranging their descriptions to meet our needs.

[1] `www.ws4d.org`

We evaluate in this section the OWL Writers in two subsections. First, in subsection 11.1.1, we outline the ontology generation time with regard to the device and service descriptions. Second, in subsection 11.1.2, we provide the description size of the generated ontologies and how much information is needed to semantically annotate the device and service descriptions.

11.1.1 Ontology Generation Time

Table 11.1: Generated Ontologies

Device Type	PC Time (seconds)	STB Time (sec)	Description (LoC[1])	OWL (LoC)
UPnP Printer	0.145	0.75	696	1573
DPWS Printer	187	763	2237	9082
UPnP Light	0.039	0.35	88	365
DPWS Light	0.8	1.1	213	245
UPnP Clock	0.05	0.1	75	161
DPWS Clock	0.41	0.85	56	123
UPnP Fake TV	0.033	0.13	73	158
DPWS Fake TV	0.32	0.72	75	115
UPnP SendPS	0.032	0.12	73	149
DPWS SendPDF	0.28	0.57	63	99

[1] Line of Code

Table 11.1 shows the time in seconds for the ontology generation per device type and the number of lines of each generated ontology (without comments and spaces). Moreover, Table 11.1 shows the UPnP or the WSDL file description used to generate the ontologies. We carried out each generation four times on a Java/OSGi framework on the PC, thus the time results in the Table 11.1 represent an average of four executions. We also added the generation time for the four fake devices descriptions used to validate the expert's code adaptation, detailed in section 9.4.4.

The difference in the building time between the UPnP and the DPWS ontologies is due to the following: first, DPWS devices have complex hierarchical parameters expressed in WSDL and XSD (see the WSDL Printer description [Microsoft 07]) while the UPnP devices have simple parameters without hierarchical structures. Thus, the ontology generation of the DPWS devices includes the WSDL flattening (see section 10.1.1). Additionally, the DPWS WSDL flattening increases the lines of code in the DPWS devices ontologies. For instance, the DPWS Printer description has 2237 lines of code, the ontology reflecting the DPWS printer contains 9000 lines of code. The UPnP Printer not having structured parameters and represented with 600 LoC description has a generated ontology of 1500 LoC.

The second element influencing the ontology generation time is the SOA4D [SOA4D b] DPWS Based Driver technical implementation choice. As mentioned in section 3.3, base drivers [UPnP , Bottaro 08b] represent real devices and services as local OSGi services. The developers of the SOA4D DPWS Base Driver [SOA4D b] chose to include in the local OSGi services only the device and service information. Thus, the local OSGi DPWS device representation ignores the operations, the parameter names and types. Their motivation is driven by the fact that clients invoking an operation on a DPWS device, previously know the operation name and parameters. Consequently, since the DPWS operations and parameters are not reified locally on the OSGi DPWS device, the DPWS OWL Writer spends an additional time to retrieve the WSDL and the related XSD files embedded from the real DPWS device. Once the WSDL is retrieved from the real device, the DPWS OWL Writer parses it and retrieves the operations and parameters names and types. Then, the DPWSL OWL Writer proceeds

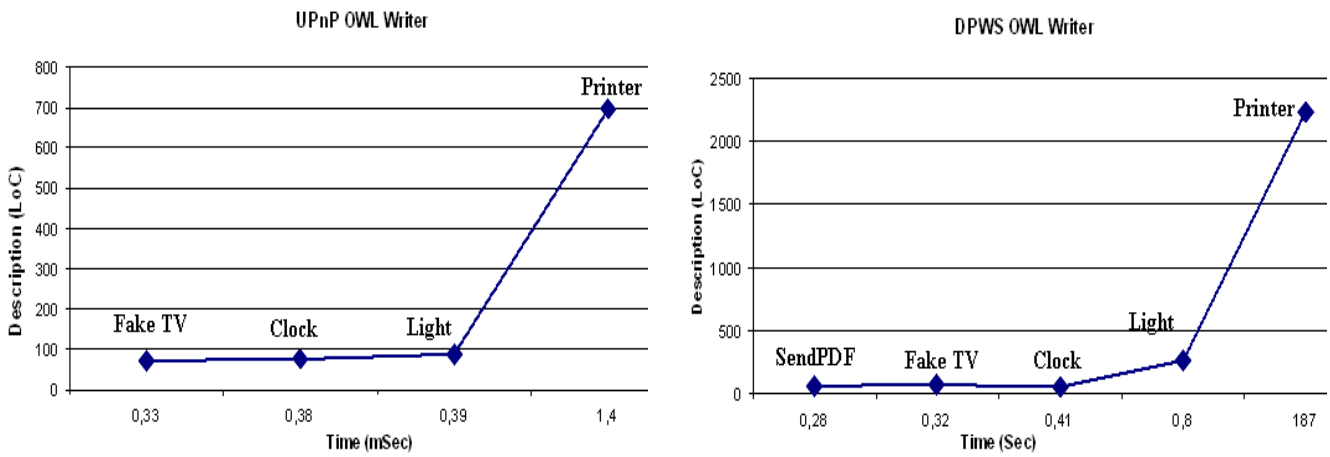


Figure 11.1: UPnP and DPWS OWL Writers Generation

with the ontology generation.

Unlike the DPWS base driver, the UPnP base driver represents on OSGi, the local devices with all the information including the action names and parameter types. Therefore, the UPnP OWL Writer has access to the needed information on the local OSGi service and there is no need to retrieve the descriptions from the real device.

Thus, an enhanced performance of the DPWS ontology generation can be achieved by improving the DPWS Base Driver in order to retrieve the complete description from the DPWS device and represent it locally on the OSGi device. Then, the DPWS OWL Writer can retrieve the required information from the base driver locally.

Table 11.1 also shows the ontology generation time on the *SodaVille* Set-Top-Box. The evaluation on the STB aims to prove that the OWL Writers can be deployed on such devices and can generate ontologies automatically in a reasonable time for devices with complex descriptions like the DPWS Printer. The difference in the generation time between the PC and the STB can be reduced probably by extending the STB RAM capacity.

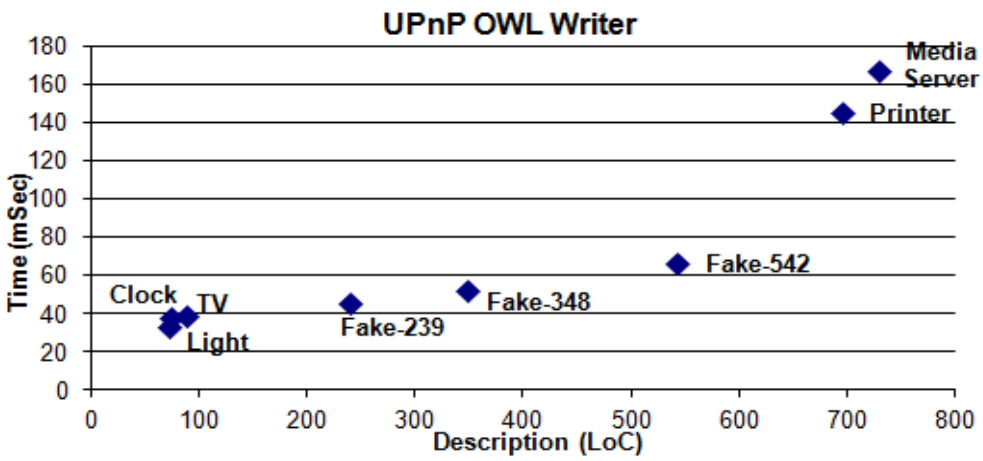


Figure 11.2: UPnP Writer Generation

Figure 11.1 plots the evaluation of Table 11.1 with the five devices [2]. From the first observation, one might concludes that the ontology generation time is exponential with regard to the description (LoC). However, in order to have a clear insight on the OWL Writers evaluation and the impact of the description size on the generation time, we used fake devices with various fake descriptions, as shown in Figure 11.2. We assigned a label to each fake device to indicate the description size in LoC[3]. Figure 11.2 reveals that the ontology generation time tends to be exponential with regard to the device description (LoC).

Table 11.2: UPnP OWL Writer Evaluation of three size equivalent descriptions

Device	Description (LoC)	Time (mSec)	Comments
Fake1	542	54.5	45 actions, 1 argument each
Fake2	542	56	16 actions, 16 arguments each
Fake3	542	62.5	32 actions with various arguments

Thus, the description size can give an idea on the ontology generation time, however other elements can influence the ontology generation performance. Therefore, we generated three UPnP fake devices with the same descriptions size as shown in Table 11.2, The *Fake1* device has 45 actions with one argument for each. The *Fake2* device has 16 actions with 16 arguments each. And finally, the *Fake3* device has 32 actions with various number of arguments ranging from 1 to 10. The ontology generation time, as shown in Table 11.2, seems to be dependent on the number of actions and arguments. The *Fake3* device having 32 actions with various number of arguments has the highest generation time. Thus, we can conclude that having a large number of actions with a multitude of arguments influences the generation time more than a device having a large number of actions with only one argument or a small number of actions with high number of arguments. However, the time difference seems to be acceptable and still fit in the exponential curve of the UPnP OWL Writer time generation shown in Figure 11.2.

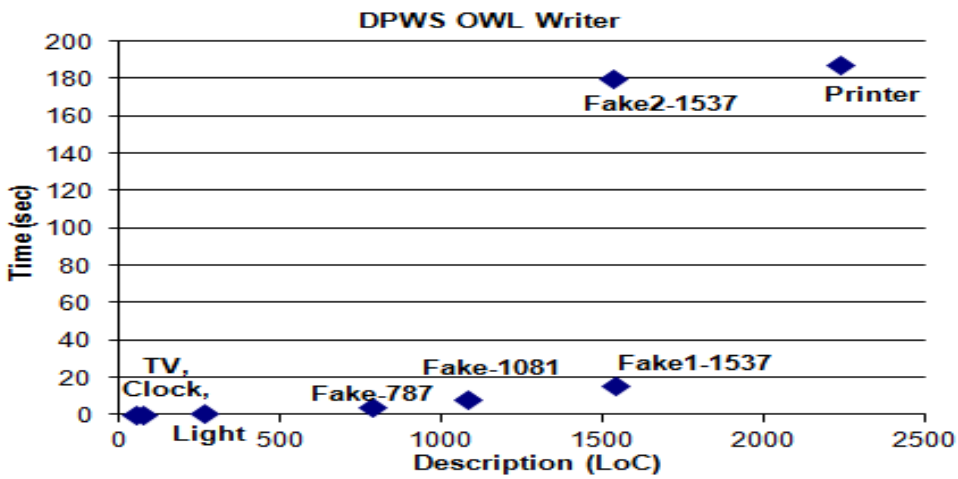


Figure 11.3: DPWS Writer Generation

Figure 11.3 shows the DPWS OWL Writer generation time with regard to the description. We also used fake devices with fake descriptions, where each fake device is labeled on the Figure 11.3. The first three fake devices

[2]The left sub-figure 11.1 shows only 4 devices since the UPnP Fake TV and UPnP SendPS have similar results
[3]We also generated the ontology of a Microsoft UPnP Media server device

have various description sizes. The *Fake1-787* device has a description of 787 LoC with 30 actions having one simple flat argument each. The *Fake2-1081* has 60 actions with one simple flat argument for each action and *Fake3-1537* has 90 actions with one simple argument for each action. The DPWS OWL Writer performance for these three devices seems to be linear according to the Figure 11.3.

However, the ontology generation of the device *Fake4-1537* shows a different behavior. In fact, the device *Fake-1537* has also a description of 1537 LoC containing 35 actions with one simple argument each and 25 actions with a *CreatePrintJob* complex argument each. The *CreatePrintJob* is a complex and structured argument with a maximum depth of 6 other structural and complex parameters, as shown in Figure B.1 in the Appendix B.2. Thus, as shown in Figure 11.3, even though the devices *Fake3-1537* and *Fake4-1537* have the same description size, the generation performance is not equivalent. This difference in the generation time is due to the complex parameters composition in the WSDL where an XSD file allows to build several complex and structural parameters. Thus, the WSDL flattening, explained in section 10.1.1, increases the generation time.

Thus, the DPWS ontology generation depends on the complexity of the structural parameters in a description. On the contrary the UPnP ontology generation follows an exponential generation time with regard to the description size. Moreover, the UPnP description is less complex than the WSDL description which allows a structural parameters compositions.

We evaluate next the description ratio of the generated ontologies with regard to the device and service descriptions.

11.1.2 Annotated Information in the Generated Ontologies

In order to evaluate the ratio of the added information in the generated ontologies, we measured the coefficient of the generated ontology size to the description size used as an input, $Overhead = \frac{\text{Generated Ontology}_{\text{LoC}}}{\text{Description}_{\text{LoC}}}$.

Thus, for the UPnP OWL Writer, we applied the overhead on each device of Figure 11.2, the obtained average $\overline{\text{overhead}}_{UPnP}$ reaches 2.7 in a $[2.04, 4.14]$ interval. As for the DPWS OWL Writer, we applied the overhead on each device of Figure 11.3, the obtained average $\overline{\text{overhead}}_{DPWS}$ reaches 1.9 in a $[1.03, 4.05]$ interval.

The $\overline{\text{overhead}}_{DPWS}$ value is less then the $\overline{\text{overhead}}_{UPnP}$. This is due to the fact that the WSDL description contains redundant and extra information not needed for the ontology generation. In fact, a WSDL description file contains several parts, mainly the *portType* part which defines a set of abstract operations (actions) and then refers to the input/output messages of each operation. Additionally, the *binding part* also contains information about the operation and specifies the protocol and data format along with the messages. Therefore, for the DPWS ontology generation, only a part of this redundant information is extracted and used.

As for the *Fake4-1537* and *Fake3-1537* which have equivalent description WSDL size, we noticed that the overhead of the *Fake4-1537* reaches 2.7, while the overhead of *Fake3-1537* has a value of 1.03. The difference in the overhead is due to the complex structural elements in the *Fake4-1537* device while the *Fake3-1537* only uses simple arguments. Thus, the WSDL flattening of complex parameters during the DPWS ontology generation, see section 10.1.1, increases the ontology description and generation time.

Table 11.3 resumes the number of generated lines of code for each ontology. It expresses the number of lines in the ontology generated for each entity type. For instance, a UPnP device entity generates 9 LoC and 6 additional lines for each supported service. Moreover, both ontology writers first generate common declarations in the ontology to specify the basic types and their associations. For example, a device is related to a service using the *hasService* relation. The difference in the templates size is due to the fact that in UPnP, we represent in the ontology the UPnP Service ID, version and name. Such information is used later during the code generation, the service lookup and binding. The additional UPnP service annotations also explains the difference in the

Entity Type	UPnP (LoC)	DPWS (LoC)
Template	42	32
Device	9+(6 per Service)	
Service	21+(6 per Action)	3+(6 per Action)
Action	3+(6 per StateVar.)	
StateVar.	Per Type	
String	6	
Boolean	14	
Enumerated Value	5	
integer	21	

Table 11.3: Ontology Generation (LoC) per entity type

generated lines of codes between the services in Table 11.3. As for the state variable, the generated lines of code in the ontology depend on the variable type, Table 11.3 gives an outline of four types.

The next section details the evaluation of the device matching.

11.2 Device Matching

We provide in this section an evaluation of the SMOA++ method on simple examples to reveal its advantages and drawbacks. Then we provide the ontology alignment results applied on five equivalent devices.

11.2.1 SMOA++

We provide in Table 11.4 similarity values returned by some basic matching techniques applied on two strings. As explained in chapter 8, some techniques return a normalized dissimilarity, $\overline{\delta}$. The relation between a normalized dissimilarity and a normalized similarity is obtained as follows $\overline{\delta} = 1 - \delta$.

Strings \ Method	1- Ngrams	1-Jaro	1-Hamming	1-Levenshtein	1-SMOA	SMOA++
(DecrementVolume, IncrementVolume)	0.84	0.93	0.86	0.86	0.92	0.0
(Particle, Article)	0.9	0.95	0.0	0.87	0.96	0.0
(SetVolumeUp, SetVolumeDown)	0.7	0.83	0.69	0.69	0.88	0.0
(Clock, Timer)[3]	0.0	0.0	0.0	0.0	0.0	1.0
(SetClock, SetTime)	0.18	0.64	0.37	0.37	0.37	1.0
(RetrieveData, FindData)	0.25	0.39	0.0	0.41	0.52	1.0
(SelectTV, SelectTelevision)	0.6	0.81	0.43	0.5	0.83	1.0
(SetVolumeUp, IncrementVolume) [1]	0.45	0.54	0.0	0.4	0.69	0.4
(SetVolume, GetVolume) [2]	0.85	0.92	0.88	0.88	0.93	0.5
(PrinterStatus, PrinterState)	0.85	0.94	0.84	0.84	0.95	0.0

[1] "Increment" and "Up" not found in WordNet as synonyms.

[2] "Set" and "Get" not found in WordNet as antonyms.

[3] Found on the UPnP and DPWS Clocks devices.

Table 11.4: Comparison between basic Matching Techniques

We expose in Table 11.4, three categories of strings. In the first three lines, the first category contains antonyms or non related pairs of strings. The second category compares 4 semantically equivalent pairs of strings. In the final category we chose specific pairs of strings to reveal the drawbacks of the SMOA++ technique.

In the first category, our proposed SMOA++ technique, based on the WordNet dictionary detects the antonyms and the non equivalent strings. The substrings *Decrement* and *Increment*, *Up* and *Donw* exist in WordNet and they can identified as antonyms. Thus, in SMOA++ (see equation 9.2) when at least one antonym is detected, the whole similarity is set to zero. The strings *article* and *particle* are not related in WordNet thus, the returned similarity is zero.

The second category, contains synonyms, SMOA++ is able to detect a valid equivalence between the strings pairs. SMAO++ is able to detect noun abbreviations between two strings like *SetTV* and *SetTelevision*. Such abbreviation exists for some nouns in WordNet.

The final category reveals two weakness points in the SMOA++ technique. The first point appears when specific domain antonyms and synonyms are not present in WordNet. In Table 11.4, the first pair of strings *SetVolumeUp* and *IncrementVolume* can be equivalent semantically, however, since the strings *Increment* is not related to the string *Up* in WordNet, then the detected similarity is not strong. Moreover, the *SetVolume* and *GetVolume* are not detected as antonyms since the strings *Set* and *Get* are not related in the dictionary. Thus, since the SMOA++ similarity is highly dependable on the WordNet dictionary, missing or non related information in WordNet influences the SMOA++ matching result. The chosen strings in Table 11.4 are specific to the device domain therefore they are not related in WordNet which contains general synonyms relations and antonyms.

The second weakness point resides in the tokenization of strings according to their existence in WordNet. Moreover, SMOA++ keeps the longest common substring found in WordNet and repeats the operation until no substrings can be identified. For example, the *CreateJob* string is tokenized into *Create*, *eat* and *Job*. The common longest substring between *Create* and *eat* is *Create* thus the tokenization result of *CreateJob* is *Create* and *Job*. However, the tokenization is not always accurate as shown in the last pair of strings of Table 11.4. The *PrinterStatus* is tokenized into *Printer* and *Status*. However, the tokenization of the string *PrinterState* identifies the following substrings: *Printer*, *State* and *interState* since the three subtsrings exist in WordNet. SMOA++ retains the substring *interState* which has the longest common substring. Then, the SMOA++ matching applied on {*Printer*, *Status*} and {*interState*} returns zero.

Thus, even though the SMOA++ detects antonyms and synonyms, a specific device domain dictionary can be used to enhance the matching. The tokenization step can also be improved using other metrics to reduce the tokenization errors.

The next section details the matching between the automatically generated ontologies representing the devices descriptions.

11.2.2 Alignment

In order to test the alignment, we took existing devices descriptions and applied our matching strategies. We applied the alignment on 3 types of devices. The *intel* UPnP Light [Apache b] and the DPWS SOA4D [SOA4D b] Light, the UPnP Felix [Apache b] Clock and the WS4D[4] Clock, and finally on a DPWS and UPnP standard printers profiles. We tested the alignment on different devices using SMOA and SMOA++. We chose the SMOA [Stoilos 05] technique since it is the most adapted to the entity name selection in the plug and play domain descriptions. As mentioned in chapter 8, SMAO is based on how programmers assign names to variables.

[4] www.ws4d.org

Table 11.5: Mapping between a DPWS and a UPnP light device

UPnP DPWS Light devices ($\overline{t_P}$)			SMOA++		SMOA	
Type	UPnP	DPWS	t=0.63	t=0.25	t=0.63	t=0.25
Device	BinaryLight	SimpleLight	–	0.5	–	0.57
Service	SwitchPower	SwitchPower	1	1	1	1
Action	GetStatus	GetStatus	1	1	1	1
StateVariable	Status	Power	–	–	–	–
Action	SetTarget	Switch	–	–	–	–
StateVariable	Target	Power	–	–	–	–
Instance	true	ON	–	–	–	–
Instance	false	OFF	–	–	–	–
Service	DimmingService	Dimming	0.8	0.8	0.9	0.9
Action	SetLoadLevelTarget	SetLevel	0.67	0.67	0.88	0.88
StateVariable	LoadLevelTarget	LightLevelTarget	0.67	0.67	0.82	0.82
Action	GetLoadLevelStatus	GetLevel	0.67	0.67	0.88	0.88
StateVariable	LoadLevelStatus	LightLevel	–	0.4	–	0.55
False Match						
Action	GetMinLevel	GetLevel	0.8	0.8	0.94	0.94
	SetTarget	TargetReached	–	–	0.67	0.67
	SetTarget	SetLevel	–	0.5	–	–
StateVariable	Target	LightLevelTarget	–	0.5	0.77	0.77
	MinLevel	LightLevel	–	–	0.67	0.67
Summary						
Success			6/12	8/12	6/12	8/12
Percentage			50%	66%	50%	66%
False Matching			1	3	4	4

Table 11.5 shows the UPnP and DPWS Lights [Apache b],[SOA4D b]) alignment similarity values using SMOA and SMOA++. The table shows the detected mappings between the two ontologies. The entities are represented in the table according to their relation with the other entities. For example, the action *SetTarget* is attached to the service *SwitchPower*. The action has the *Target* entity as a variable which has two instances *true* and *false*. The "–" symbol refers to undetected mappings. Table 11.5 also shows the false detected matching and outlines the percentage success % of each method using two different threshold values. The success rate represents the number of valid detected mappings divided by all the valid mappings. We applied two hard thresholds t=0.63 and t=0.25, i.e. all the alignment tuples having similarity values higher than the threshold t value are retained. Table 11.5 shows the results without applying a similarity propagation between the entities, i.e $\overline{t_P}$.

The evaluation result shows that SMOA and SMOA++ have the same success rates 50% and 66% on the lights ontologies. However, the SMOA++ has clearly lower false matches when a higher trimming threshold value is applied. The number of false matches goes from 3 to only 1 with SMOA++. While in SMOA, the number of false matches remains the same.

The Table 11.6 shows the alignment results with the similarity propagation compared with the previous table where no propagation is applied. The propagation expressed in equation 9.4, allows to enhance the similarity value between two entities if their children have a similarity value equal or higher than the threshold value t_P. We show in Table 11.6 the similarity propagation with the following threshold values: $t_P = 0.4$, $t_P = 0.8$, and $t_P = 1$. A hard trimming threshold is applied with $t = 0.63$.

Table 11.6: Mapping between a DPWS and a UPnP light device with Similarity Propagation

UPnP DPWS Light devices			SMOA++			
Type	UPnP	DPWS	t_P=1	t_P=0.8	t_P=0.4	$\overline{t_P}$
Device	BinaryLight	SimpleLight	0.75	0.7	0.69	0.5
Service	SwitchPower	SwitchPower	1	1	1	1
Action	GetStatus	GetStatus	1	1	1	1
StateVariable	Status	Power	–	–	–	–
Action	SetTarget	Switch	–	–	–	–
StateVariable	Target	Power	–	–	–	–
Instance	true	ON	–	–	–	–
Instance	false	OFF	–	–	–	–
Service	DimmingService	Dimming	0.8	0.8	0.77	0.8
Action	SetLoadLevelTarget	SetLevel	0.67	0.67	0.67	0.67
StateVariable	LoadLevelTarget	LightLevelTarget	0.67	0.67	0.67	0.67
Action	GetLoadLevelStatus	GetLevel	0.67	0.67	0.53	0.67
StateVariable	LoadLevelStatus	LightLevel	0.4	0.4	0.4	0.4
False Match						
Action	GetMinLevel	GetLevel	0.8	0.8	0.8	0.8
	SetTarget	SetLevel	0.5	0.5	0.5	0.5
StateVariable	Target	LightLevelTarget	0.5	0.5	0.5	0.5
Summary						
Success with ($t = 0.63$)			7/12	7/12	6/12	6/12
Percentage with ($t = 0.63$)			58%	58%	50%	50%
False Matching ($t = 0.63$)			1	1	1	1

The similarity propagation results shown in Table 11.6 reveal an improvement in the success percentage when the similarity propagation threshold t_P is equal or higher than 0.8. Additionally, the number of false matches remains the same. Thus, the similarity propagation allows to improve the matching result when used with an adequate selected threshold.

Table 11.7 shows the alignment results applied on the UPnP and DPWS clocks. We apply SMOA and SMOA++ with two threshold values $t = 0.63$ and $t = 0.25$. The results of Table 11.7 do not include a similarity propagation. The SMOA++ based on the WordNet dictionary is able to detect a similarity between the two services, the *SimpleClockService* and the *timer*. Thus, the percentage success of SMAO++ with a threshold $t = 0.25$ reaches 66%. Then, it is up to the epxert to add the 2 remaining matchings uwing ATOPAI to attend a successful mapping between the two clocks.

We show in Table 11.8 the alignment results with a similarity propagation method. The results are improved with two propagation similarity thresholds $t_P = 0.4$ and $t_P = 0.8$.

Table 11.7: Mapping between a DPWS and a UPnP Clock device

UPnP DPWS Clock devices ($\overline{t_P}$)			SMOA++		SMOA	
Type	UPnP	DPWS	t=0.63	t=0.25	t=0.63	t=0.25
Device	clock	SimpleClock	0.67	0.67	0.81	0.81
Service	timer	SimpleClockService	–	0.5	–	–
Action	GetTime	GetCurrentTime	0.8	0.8	0.71	0.71
	SetTime	SetCurrentTime	0.8	0.8	0.71	0.71
StateVariable	Time	Input	–	–	–	–
	Time	Result	–	–	–	–
False Match						
StateVariable	Result	Result	1	1	1	1
Summary						
Success			3/6	4/6	3/6	3/6
Percentage			50%	66%	50%	50%
False Matching			1	1	1	1

Table 11.8: Mapping between a DPWS and a UPnP Clock device with Similarity Propagation

UPnP DPWS Clock devices			SMOA++			
Type	UPnP	DPWS	t_P=1	t_P=0.8	t_P=0.4	$\overline{t_P}$
Device	clock	SimpleClock	0.67	0.67	0.66	0.67
Service	timer	SimpleClockService	0.5	0.65	0.65	0.5
Action	GetTime	GetCurrentTime	0.8	0.8	0.8	0.8
	SetTime	SetCurrentTime	0.8	0.8	0.8	0.8
StateVariable	Time	Input	–	–	–	–
	Time	Result	–	–	–	–
False Match						
StateVariable	Result	Result	1	1	1	1
Summary						
Success with ($t = 0.63$)			3/6	4/6	4/6	3/6
Percentage with ($t = 0.63$)			50%	66%	66%	50%
False Matching ($t = 0.63$)			1	1	1	1

The alignment evaluation on relatively two simple device types lights and clocks revealed high success matching rates reaching up to 66%. A high value hard trimming threshold allows to reduce the errors and remove the false matchings. In Table 11.5, with SMOA++, the number of false matching is reduced from 3 to only 1 with a hard trimming threshold value $t = 0.63$. However, a high level trimming threshold value also eliminates valid matchings having a weak similarity. Thus, an alternative solution consists in applying a high value hard trimming threshold t with a high level propagation similarity threshold t_P to enhance the similarity values of entities based on the similarity of their mapped childs.

We expose next the alignment results between the two standard printers ontologies. The UPnP and DPWS printers represent the most complex standardized devices so far in the UPnP Forum and the DPWS standard profiles. First, we expose in Table C.1 in the appendix, the list of symbols used. Each alignment also indicates the success and false matching percentage rates for two hard trimming thresholds $t = 0.63$ and $t = 0.25$.

Tables C.2, C.3 and C.4 in the Appendix C show the alignment results of the aligned printers using the SMOA matching technique. The SMOA based alignment detects 20 out of 28 valid matching along with 6 false matches when a hard trimming threshold $t = 0.63$ is applied. With a lower hard trimming threshold value $t = 0.25$ the number of false matches highly increases to attend 11 with only one additional valid matching.

The SMOA++ alignment results are expressed in Tables C.5, C.6 and C.7. The propagation similarity is not applied. The success rate of SMOA++ achieves 78% with a hard trimming threshold value of $t = 0.25$. SMOA++ also detects 8 false matches. However, with a high trimming threshold $t = 0.63$, SMOA++ achieves a lower success rate of 71% with only two false matches.

We also detail in Table C.8, the SMOA++ alignment results assisted with a propagation similarity with three threshold values $t_P = 0.4$, $t_P = 0.8$ and $t_P = 0.1$. The alignment results are represented in Table C.8 without the state variables similarities. Since the similarity propagation applies a down-top enhancement, the state variables are taken into account to enhance the similarities of the actions. Thus, the similarity values of the state variables don't vary when the similarity propagation is applied. In Table C.8, the similarity propagation allows to enhance the similarity value of (*PrinterEnhancer*, *PrinterService*) and the *GetPrinterAttributes* union. Thus, the similarity propagation increases the success rate of the alignment results.

The following Table 11.9 resumes the alignment evaluation on the three devices (UPnP/DPWS) using SMOA and SMOA++. It exposes the methods used, the alignment time is seconds, the success percentage for the hard trimming thresholds $t = 0.63$ and $t = 0.25$ as well as the number of false matches found. Both techniques have similar success rates when the trimming threshold is set to $t = 0.63$. However, the number of false matches detected in SMOA++ is much lower than those detected in SMAO in all the alignments.

Table 11.9 shows that SMOA++ spends more time than the SMOA method in all the alignment results. In SMAO++, the Tokenization step relies on the WordNet to split a string into several existing substrings in the dictionary. More over, the matching step in SMOA++ relies on the synonyms, antonyms and coordinates search in WordNet. Thus, the huge difference in time is due to the search in WordNet. However, such difference in time remains acceptable since the alignment is treated off line.

SMOA++ provides better results with a trimming threshold value set to $t = 0.25$. The success rates achieve 78%, however the number of false matches increases from 2 to 8 on the printers alignment for example. SMOA achieves a maximum success rate of 75% on the printers, however, the number of false matches increases from 6 to 11. In the other alignments, SMOA++ success rates are equal or higher than the SMOA's rates and the number of false detected correspondences are equal or lower.

The hard trimming threshold values are set by the expert using ATOPAI. A low threshold value returns high success rates, however it also retains a high number of the false detected matches. Thus, to reduce the number of false detected matches, a higher trimming threshold value is used, since it will remove all the weak similarities lower than the threshold value. However, a higher trimming threshold value also reduces the successful detected

Type	Method	Time (sec)	success(t1=0.63)	fails(t1=0.63)	success(t2=0.25)	fails(t2=0.25)
Printers	SMOA++	824	71%	2	**78%**	8
	SMOA	77	71%	6	75%	**11**
Lights	SMOA++	7.3	50%	1	**66%**	3
	SMOA	1.5	50%	4	**66%**	**4**
Clocks	SMOA++	2.5	50%	1	**66%**	1
	SMOA	0.4	50%	1	50%	1

Table 11.9: Alignment Evaluation without Similarity Propagation

matchings. A compromise can be achieved using the similarity propagation on a high trimming threshold value. The similarity propagation enhances the similarity value of two entities if the similarity values of their matched childes are higher than the propagation similarity threshold value t_P. Figure 11.4 summarizes the similarity propagation results applied on the alignments with a trimming threshold value $t = 0.63$. For each alignment, Figure 11.4 shows the success rates when no similarity propagation is applied (SimProp[5]=None) and the success rates when three different propagation similarity values are set, SimProp=0.4, SimProp=0.8 and SimProp=1.

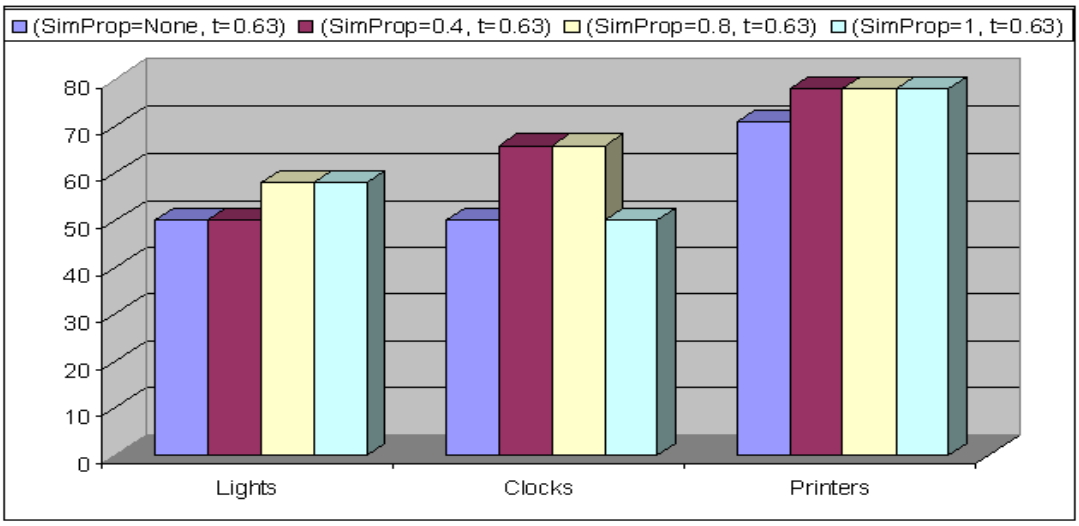


Figure 11.4: Success rates with regard to several similarity propagation values

Figure 11.4 shows an enhancement in the lights success rate with two similarity propagation thresholds (SimProp=0.8 and SimProp=1). The clocks success rates increase with a similarity propagation thresholds (SimProp=0.4 and SimProp=8) while the printers success rates increase starting from a similarity propagation threshold SimProp=0.4. Thus, on these three alignments, a propagation similarity threshold value in the following range [0.4, 0.8] allows to enhance the similarity values and to increase the success rates. Additionally, as shown in Tables 11.6, 11.8, C.8 the number of the false matchings did not increase.

Based on these results, we recommend the expert to set a hard trimming threshold value $t = 0.63$ and a similarity propagation $t_P \in [0.4,\ 0.8]$ using ATOPAI for future device matching. Obviously, the recommended values can be re-tuned based on the new alignment results in order to achieve a combination of a hard trimming and a similarity propagation thresholds. Such combination aims to increase the number of successful detected mappings and to reduce the retained false matchings. Once the expert has validated the alignments, using

[5]SimProp is t_P, We used SimProp in the Figure 11.4 for more clarity

Device Type	Printers	Lights	Clocks
Time (sec)	23.2	2.5	1.8

Table 11.10: Patterns and Matching Concepts Detection Time

ATOPAI he triggers the pattern and the matching concepts detection steps. Table 11.10 shows the time in ms for the pattern detection and classification. The 12 rules are applied on the three ontologies and then the matching concepts classification follows. The pattern and the matching concept detection depend on the size and the complexity of the ontology. Thus, the time spent on the clocks and the lights is lower time than the time spent on the printers.

Table 11.11: Device Matching: number of the matched entities

Device	UPnP	DPWS	Total Entities	Matched Entities
Printer	33	71[1]	104	60
Light	13	17	30	24
Clock	6	6	12	12
TV	5	6	11	11
PS	4	4	8	8

[1] For the variables, we counted the various leaves in the complex parameters.

Table 11.11 outlines the number of entities in each device description (UPnP and DPWS). An entity is a device, service, action or a variable. The table also shows the number of matched entities and retained in the ontology alignment. The matched entities are used by DOXEN to generate the adaptation code. For instance, there is 60 matched entities to be used by DOXEN during the printer proxy generation. The rest of the entities are ignored since there is no equivalent correspondences relating in between.

Device	UPnP	DPWS	Before Align.	Final Ontology	Additional Annotation
Printer	1573	9082	10655	11255	600
Light	365	245	610	706	96
Clock	161	123	284	368	84
PS	149	99	248	311	63
TV	158	115	273	304	31

Table 11.12: Device Matching: description size increase

The aligned entities for the five devices is greater than 57% (60/104). The detected entities influence the size of the final ontology. In fact, Table 11.12 shows the description size of each generated ontology, the size before and after the ontology alignment. For instance, the printers ontology before the alignment reaches 10655 LoC. However, after the alignment and validation, the final ontology reaches 11255 LoC, thus, the alignment added 600 LoC. The added information represents the alignment annotation between entities from both ontologies along with the pattern annotation.

During the code generation, DOXEN generates bundles based on the matched entities and the added annotations. DOXEN ignores the entities not having any relation with other entities.

The next section details the evaluation of the DOXEN module, the time spent to generate and start a proxy on the PC and the Set-Top-Box.

11.3 DOXEN

In this section, we expose the proxy generation evaluation time, the number of generated files and size of the packaged bundles. We also outline the invocation overhead between an application and a DPWS device receiving invocations through the proxy.

11.3.1 Proxy Generation

Table 11.13 resumes the time spent by DOXEN to (1) generate Java Code, (2) compile it, (3) package it in Java/OSGi Jars and finally (4) install it. We also show the number of Java template files and lines of code (LoC) used (without spaces and comments) as well as the number of automatically generated Java Files and their LoC. The table also shows the cost of adding a basic and a configuration management services. The proxy supporting the two management services (BMS, CMS) is referred to as an Manageable Device (MD) [UPnP 11]. The values shown in the tables are an average result of three executions. We also show the proxy generation of the two use cases, the fake TVs and the PS-to-PDF conversion described in section 9.4.4.

Proxy	Time (sec)		Java files		Lines of Code		Jar (KB)	
		MD		MD		MD		MD
Templates	–	–	8	–	1112	–	–	–
MD: BMS+CMS	–	–	–	49	–	5651	–	–
Printer	1.7	4.43	31	80	3325	8976	67	163
Light	0.94	2.06	15	64	1812	7463	37	133
Clock	0.85	2.01	9	58	1151	6802	22	116
UC#1 [TVs]	0.72	1.76	9	58	1198	6849	25	119
UC#2 [Ps-to-PDF]	0.77	1.98	9	58	1194	6845	25	119

Table 11.13: Generated Proxy results on a PC

DOXEN uses 8 Java template files to generate the three proxies. It generates the printer UPnP-DPWS proxy in 1.7 seconds on the PC, which contains 31 generated Java files. The size of the Jar reaches 67 KB, which might be reduced if another compiler is used instead. Table 11.13 also shows the time to generate a printer MD proxy supporting the two management services. The generation time increases to attend 4.43 seconds on the PC. The number of generated Java files reaches 80 and the jar size also increases.

We also deployed the DOXEN module on a SodaVille Set-Top-Box (STB) with an Intel Atom 1.2 GHz, (256+128) MB of RAM and an open-JDK 6 implementation. The generation time, shown in Table 11.14, ranges from 6 to 16 seconds. The time spent on the STB is higher than the generation time on the PC probably due to the lower RAM capacity on the STB. The time spent on the STB and the PC reveal that the DOXEN module can be deployed in the digital home to generate proxies on the fly in a reasonable time.

We detail next, the invocation overhead going through the proxy between a UPnP application and a DPWS device.

11.3.2 Proxy Invocation

We implemented DPWS specific control points to evaluate the invocation overhead through the generated proxies. We also extended the UPnP Felix Control Point to measure the action invocation time. The UPnP Control point and DPWS applications along with the generated proxies are deployed on the same PC and

Proxy	Time (sec)	Time (MD)(sec)
Printer	10.1	16.4
Light	4.7	8.18
Clock	3.7	6.5
UC#1 [TVs]	3.4	6.02
UC#2 [Ps-to-PDF]	3.8	6.8

Table 11.14: Generated Proxy results on a Sodaville STB

connected through the local network to the printer and other PCs hosting the DPWS devices. The evaluation are measured on the UPnP control point and the DPWS applications as shown in Figure 11.5.

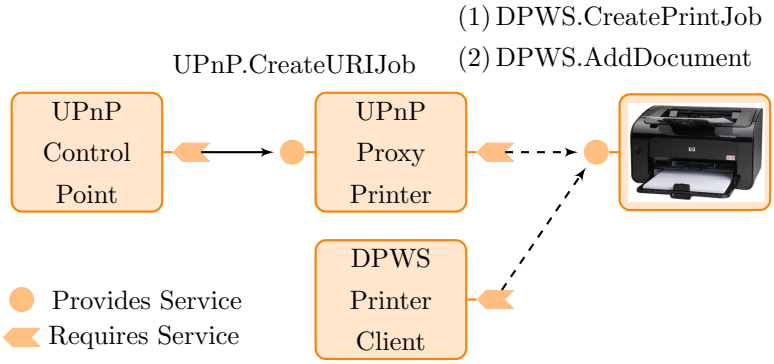


Figure 11.5: Real Printer Example

Table 11.15 shows the invocation time of a DPWS Client invoking directly the actions on the HP 4515x Printer. The table also reveals the invocation time of the UPnP control point going through the generated UPnP Printer Proxy (see Figure 11.5). The clients printed the same file size and used three different formats (pdf, ps, txt). The results show a small difference which represent the time of redirecting the action invocation, and the WSDL structure generation along with the parameter filling with the appropriate UPnP received values. The *CreatePrintJob* input structure is shown in Figure B.1 in the appendix B.2. A generated WSDL structure is used as an input parameter when invoking the DPWS actions. The DPWS actions also return a WSDL structure containing values to be returned to the UPnP application, such as the JobId, or the job attributes. Thus, the proxy parses the returned WSDL structure and retrieves the values, then translates them into UPnP compatible values and returns them to the UPnP application. The difference in time between the two clients ranges from 17 ms to 297 ms. Such difference in time is insignificant to the user or the application invoking the actions. We also measured the time difference for the other devices and proxies, the difference in time is less than 5 ms.

		Invocation Time in ms	
UPnP Action	DPWS Actions	UPnP CP	DPWS Client
CreateURIJob	Sequential Union (CreatePrintJob, SendDocument)	1141	844
Cancel Job	CancelJob	57	40
GetPrinterAttributes	Union (GetPrinterElements, GetActiveJobs)	583	357
GetJobAttributes	GetJobElements	481	364

Table 11.15: Printer action invocation Time (ms)

11.4 Discussion

We propose in this work to resolve the heterogeneity between plug and play devices by applying ontology matching techniques to find the correspondences between two equivalent devices. As mentioned in chapter 4, there is three levels of heterogeneity between the plug and play devices. The first level of heterogeneity resides in the protocol stacks, the second level exists in the description format and the third and final level is in the description content. The protocols stacks heterogeneity is handled by the OSGi base drivers which represents the devices as local OSGi services. Thus, the applications can interact directly with the devices through the API base drivers methods.

In order to resolve the second and the third levels, we propose three modules. The first module consist in automatically generating ontologies based on the devices' descriptions and capabilities. The generated ontologies resolve the second level of heterogeneity since the description is now represented in a common format, the ontology format which is conform to the meta model device. Additionally, the ontology allows to express the relation between the entities, such as the sub-concept or the equivalent relation. Thus, the ontology represents each device, its services along with the actions and variables in a common representation using the ontology domain semantics. The automatic ontology generation as shown in section 11.1, varies from less than 0.05 seconds to 4 minutes when dealing with complex devices such as the printers.

To resolve the third level, we rely on the semi-automatic ontology matching techniques to resolve the content description heterogeneity. The ontology matching assisted with a semantic dictionary allows to detect similar entities independently from their syntactic name description. The semantic dictionary is used to detect the synonyms and antonyms between the entities. Section 11.2 reveals the ontology alignment evaluation, the matching can detect up to 78 % of the valid correspondances. Then, it is up to the epxert to valid, update and edit the alignments to achieve a valid ontology mapping. We also propose the ATOPAI framework along with a GUI to make the ontology validation easier. Based on the valid alignment, we apply several rules on the ontology to detect actions compositions (union or sequential) based on their input and output variables. The rules automatically annotate the ontology with the detect pattern category and composition. Then, we automatically classify the compositions and return to the expert the non-valid compositions. The expert then checks based on the devices specifications if an adapation is possible. There are three possible adaptations, the first consists in setting default input/output parameters values. The second category requires an adaptation behavior on the actions and their parameters. The third and final category allows to delegate the adaptation to external services such as OSGi services on the framework.

Once the ontology alignment is validated and contains the transformation to go from one description to another. DOXEN takes the high level representation then generates and installs a specific proxy which handles the invocations adaptations. As shown in section 11.3, the proxy generation is feasible, the DOXEN module can be deployed in the digital home to generate proxies based on the ontologies. The proxy generation ranges from less than a second and up to 16 seconds. Furthermore, the invocation overhead through the generated proxies ranges from 17 to 297 ms which can be considered inconsiderable.

We complete in Table 11.16, the comparison by adding our approach characteristics. The proposed approach can be applied to the following plug and play protocols: UPnP, DPWS, IGRS and Bonjour. Our proof of concept prototype is applied for now on the UPnP and DPWS devices. However, since the IGRS protocol relies on the same protocols stacks and uses the WSDL to announce its capabilities. Then, our approach should be applicable on the IGRS devices. We already treat the WSDL descriptions provided by the DPWS devices. Furthermore, in [Xie 09] a technical solution has been proposed to resolve the heterogeneity between the UPnP and IGRS protocol to allow the device discovery and interaction.

As for the Bonjour devices, we propose a solution in section 10.1.2, based on the *HomeSOA* framework which

exposes the Bonjour devices in a WSDL.

Thus, in Table 11.16, we specify that the base drivers in our approach allow to resolve the protocol heterogeneity by representing the device and their hosted services as local OSGi services. The OWL Writers modules resolve the representation heterogeneity by automatically generating ontologies using common representation concepts. The content mediation reconciliation is provided by the semi-automatic device matching assisted by the expert. And finally, the realization of the interoperability is handled by DOXEN and the generated proxies. In our proposed approach, the manual device and service description annotation is not required since the rules automatically detect the patterns and annotate the ontology. The human intervention is needed however to validate the device matching since it is heuristics based.

Protocols Used	Protocol Mediation	Presentation Mediation	Content Mediation	Realization	Description Annotation	Human Intervention
UPnP, DPWS, Bonjour, IGRS	Base Drivers	OWL Writers	Device Matching, Alignment Validation	DOXEN	Not Required	Alignment Validation

Table 11.16: Completing Comparison of Table 7.1

The main advantages of this approach can be resumed as follows: **First**, already written applications which targets only **standard** UPnP devices can now interact with other non-UPnP devices thanks to the dynamically generated proxy. Applications can now interact transparently with any equivalent non-UPnP device using standard UPnP service and action names since the proxy is exposed as a UPnP standard device. More over, all the code generation, compilation and installation is **automatic** and transparent to the user and the UPnP applications.

Second, there is no need to add additional networking stacks to support other protocols on devices hosting applications. The same UPnP stack already deployed is used to interact with other devices supporting a different protocol via the proxy. For instance, we only deployed on the Android Smart Phone the UPnP Home Controller and a UPnP protocol stack. The application which can only interacts with the UPnP devices is currently able to interact with a standard DPWS printer through the generated proxy.

And finally, the construction process of ontologies is relatively simpler and faster than building a global common ontology specially when dealing with complex devices like the printers. The ontologies can also be reused if another protocol is chosen as a pivot. The expert validating the alignment using ATOPAI needs only to remove or add lines between the equivalent entities, the expert may also add an adaptation behavior using a high level API. The specifications are protocol and technology independent therefore the expert performing the validation off line on the operator site using ATOPAI can be a technician or a domain expert. For the printers validation, we validated the mappings by referring to the standard printers profiles [Microsoft 07, UPnP 06a]. Furthermore, to our knowledge, we are the first to resolve heterogeneity between two standard profile printers.

11.5 Conclusion

We evaluated in this chapter our proposed solution with its three modules: the OWL Writers, the device matching and DOXEN. The evaluation of the OWL Writers proves that such modules can be deployed in the digital home, on a set top box for example. The OWL Writers are capable of generating ontologies in a reasonable time when new devices appear in the network. The ontology generation vary from few seconds up to 4 minutes when dealing with complex devices.

The device matching evaluation on complex devices can actually detect up to 78% of the valid correspondences between two ontologies. Then, it is up to the expert to validate and edit the remaining correspondences

in order to achieve a valid ontology alignment. We also pointed out the strengths and weaknesses of the SMOA++. The evaluation reveales that the SMOA++ techniques can achieve better success rates and lower false matchings than the SMOA technique. However, the SMOA++ spends more time to align than SMOA due to the access to the WordNet dictionary. The device matching can aslo rely on the similarity propagation in order to enhance the similarity values based on the structure and therefore increase the success rates of the alignment.

The evaluation of DOXEN shows that it can be deployed in the digital home on a Set-Top-Box for example to generate proxies. The proxy generation time on the Set-Top-Box ranges from 4 to 16 seconds which remains acceptable. The end-user and the application will wait at most for 16 seconds for the proxy generation before discovering the new proxy-device. Furthermore, the proxy invocation overhead can achieve at most 0.2 seconds which in inconsiderable. The proxy or the end-user will wait up to 0.2 sec before the execution of their tasks.

Part IV

Conclusion

Chapter 12

Conclusion

"Reasoning draws a conclusion, but does not make the conclusion certain, unless the mind discovers it by the path of experience."
– Roger Bacon

The digital home is heading towards a ubiquitous computing environment where devices and applications cooperate and interact to accomplish the user's daily tasks. Heterogeneous plug and play protocols cohabit in the same home network, however, the cross protocol interaction between devices and applications is hard to achieve. Such heterogeneity between protocols, prevents the ubiquitous applications to use any available device, regardless of their protocol, to accomplish a certain task such as printing or dimming lights.

The plug and play protocols share a lot in common, they all announce their presence and departure on the network, they provide a list of their supported services. Each service provides a set of operations which can be remotely invoked on the network to accomplish a specific task.

Even though, the plug and play protocols have a lot in common, three levels of heterogeneity retain the plug and play interoperability. First, each plug and play protocol relies on specific protocols stacks to announce its presence/departure, and its supported capabilities. More over, each protocol uses a different set of other protocols to support the interaction and control. Thus, this heterogeneity in the protocol stacks does not allow to UPnP applications for example, searching for a UPnP specific device to detect other equivalent plug and play devices.

The second heterogeneity level resides in the description format. Each plug and play protocol defines or uses a specific description format to announce the device description and its supported capabilities. Thus, applications targeting a specific plug and play protocol are unable to interpret other devices descriptions since they are expressed in a different description format.

The content is the final heterogeneity level. Each plug and play protocol defines a set of standard profiles which contains the capabilities to be supported by a standard device. The profile also specifies the exact names to be used in the description annunciation for the device type, services, actions and variables. Additionally, the profile defines the behavior of the device when an operation is invoked along with the input/output required parameters for a successfully invocation. Thus, two devices having the same type will express their description content in a syntactically different representation. However, the description is more or less semantically equivalent. In fact, similar devices share common functionalities, for instance a light is always expected to turn on or off a light, a clock will always return the current time and a printer is always expected to print.

Thus, this three levels of heterogeneity between the plug and play devices retain the applications to use any equivalent available device in the network to accomplish a specific task. The ubiquitous computing vision consists in an open world, where devices are aware of each others and put in common their capabilities to assist

humans in their daily tasks. The heterogeneity between plug and play protocols encloses ubiquitous applications in the same protocol area and forces already implemented applications targeting a specific protocol to stay in the same perimeter. We believe that ubiquitous applications should be set free from the technical implementation details and use any available device capable of satisfying the applications needs.

A naive solution to liberate the applications would embed in each application the required implementation to support multiple protocols. However, supporting multiple protocols for each application is time consuming and error prone for the applications' developers. More over, such solution is not easily expendable and requires a manual adaptation.

Therefore, we propose in this thesis a novel approach to provide heterogeneity between equivalent device types supporting different protocols. The approach relies on the intersection of two major domains, the ontology matching crossed with the model driven engineering. The ontology matching allows to semi-automatically detect the correspondences between equivalent devices. Then, the adaptation to support multiple protocols is fully automated and relies on the previously validated ontology alignments. The adaptation is realized by automatically generating proxies. More over, there is no need to adapt the deployed plug and play applications since they will interact with the generated proxies transparently as a standard device supporting the same protocol.

We summarize next the three contribution this thesis provides.

12.1 Contributions

The main contributions of this thesis are the following:

- **An End To End Adaptation Solution:** Our proposed approach provides an end to end solution which consists of three modules to solve the three previously identified plug and play heterogeneity levels. The protocols stacks, the description format and the description content.

  To resolve the first heterogeneity level, we rely on the technical solution provided by the service oriented framework OSGi and its base drivers. Each base driver reifies real devices as local OSGi services. Additionally, the base driver offers an API to interact with the real device on the network without going into the technical details of the protocol stacks. Thus, the base drivers refine the protocols stacks heterogeneity level into an API heterogeneity level.

  To solve the second heterogeneity level, we propose the OWL Writers which automatically generate ontologies from devices descriptions. Each ontology captures the device description and the relation between its entities. By construction, the generated ontologies are conform to meta model which guarantees a unified representation. Thus, the OWL Writers resolve the heterogeneity level between the description format by exposing the device description in an ontology. The generated ontologies are then sent to the operator's site to detect the correspondences with another equivalent UPnP device.

  The third level of heterogeneity, the description content, is handled by ATOPAI which relies on: the semi-automatic alignment techniques, the rules to detect composition patterns and on the expert intervention to validate the detected correspondences. The device matching contains the adaptation behavior between the two devices. Thus, the device matching resolves the content description heterogeneity.

  Once the alignment is completed and validated, rules are applied on the ontology to annotate it with composition patterns between the actions. We also proposed the device matching concepts to assist the expert and point out the non valid compositions. Then, the expert refers to the devices specifications to check if an adaptation is possible by adding an adaptation high level code.

And finally, the DOXEN module realizes the interoperability by generating specific proxies behaving according to the ontology alignment. The proxy generation relies on the model driven engineering techniques which allow to go from a high level description into an executable code. The high to low level transformation is specified once and used each time an adaptation is needed. The high-low level transformation relies on the template filling and code generation techniques.

- **SMOA++:** The second contribution of this thesis resides in the device matching technique. We proposed an enhanced matching technique *SMOA++* inspired from the *SMOA* [Stoilos 05] technique. *SMOA++* relies on an external dictionary WordNet [Fellbaum 98] to provide a similarity value between two entities. The *SMOA++* consists of two steps. The first is the tokenization step, where a string representing an entity name such as the service name is split into several substrings. The splitting process relies on the dictionary, each substring must exist in WordNet. The, we retain from the returned list of substrings, only the biggest common substring from those having common parts.

  The tokenization is applied on the two entities from two ontologies, thus it returns two lists of substrings. The second step is the matching between the two lists of substrings. *SMAO++* computes the similarity value between two strings based on their relation in WordNet which can be queried for the synonyms and the antonyms.

- **Prototype Validation On Real Devices:** In order to validate our approach, we implemented three independent modules. The first module contains the OWL Writers which automatically generate ontologies from the devices annunciations on the network. The evaluation of the OWL Writers show that such modules can be deployed in a home network, on a Set-Top-Box or a PC for example, to automatically generate ontologies when new devices appear.

  ATOPAI, the second module, it handles the device matching to detect correspondences between equivalent devices' ontologies. ATOPAI allows an expert to easily update or edit detected correspondences using a GUI. More over, ATOPAI allows to apply the rules on the ontology to detect the pattern compositions. Additionally, ATOPAI offers to the expert a high level adaptation API in order to add adaptation behavior between actions.

  SMOA++ is used in this device matching step, the evaluation of SMOA++ shows higher or equivalent successful correspondences than those detected by SMOA. More over, the number of false matched in *SMOA++* is equal or lower that SMOA results.

  The device matching evaluation on the three devices: clocks, lights and printers shows encouraging matching results, 78% of the successful correspondences are semi-automatically detected. Then, it is up to the expert to correct and complete the matching using A GUI as the one we propose with ATOPAI.

  The third and final module is DOXEN which takes an ontology alignment and generates a specific proxy. The generated proxy behaves according to the adaptation and compositions annotated in the ontology by the previous module.

  We successfully deployed DOXEN on a Set-Top-Box in a home network. The proxy generation time carried out by DOXEN ranges between 3 to 10 seconds. In other words, when a DPWS printer appears in the home network, the user or the application waits for 10 seconds, the required time for DOXEN to generate a proxy, install it and start it. Moreover, the generation has to be done only once: when the new device is firstly discovered. Furthermore, the generation is quite fast, compared to the time spent on the implementation process carried out by a human. Additionally, to our knowledge, we are the first to print, cancel a job, retrieve the printer's and the job's attributes on a DPWS printer from a UPnP control point.

In fact, we are even the first to generate ontology-based proxies to resolve the heterogeneity between two equivalent devices.

More over, we showed in a basic diagnostic use case the benefits of supporting two management services (BMS and CMS) on each generated proxy and how such services can be used to detect basic connectivity failures.

The next section overviews the perspectives of this thesis.

12.2 Perspectives

This thesis proposes two kinds of perspectives, the first category containing the first three perspectives is related to the adaptation process, while the second category holding the last two aims to technically improve our prototype.

12.2.1 Machine Learning Based Alignment

ATOPAI allows to store a validated alignment. It actually stores the tuples (leftEntity, similarity, rightEntity) in a database to be reused later to enhance current computed alignments. In fact, to enhance the current alignment, ATOPAI queries the database searching for a tuple having the same left and right entities with a higher stored similarity value. If an equivalent stored tuple is found with a higher similarity value, then, the current similarity value between the tuples is replaced with the similarity from the database.

The current enhancement method is basic and cannot differentiate between absolute correct correspondences and device specific correspondences which are relative correspondences. For instance, the two strings *timer* and *clock* are absolute correct correspondences while *SetTarget* and *Switch* are relative correspondences which are only valid on the lights devices.

The absolute correspondences are usually found in WordNet while the relative correspondences are set by the expert validating the ontology. Thus, it would be relevant to take into account the relative and absolute correspondences when enhancing and even when applying the matching techniques.

The absolute and relative correspondences constitute only a small step to enhance the ontology alignment. More generally, the ontology alignment can rely on the machine learning to detect correspondences. Machine learning algorithms can be used to observe and learn from the expert when he intervenes for validation. Such algorithms can rely on the previous used metrics and methods to compare and analyze the correction made by the expert. Consider for example, that the expert removed a matching between *SetTarget* and *GetTarget*. Then, the machine learning can apply the tokenization step used in SMAO++ which returns the following substrings: {*Set*, *Target*} and {*Get*, *Target*}. And then, the machine learning algorithm discovers that *Set* and *Get* can be probably antonyms.

Furthermore, the machine learning can rely on statistics to take decisions during the alignment. Consider for example, that the same two strings were considered only twice as correct matchings and 10 times as false detected matching. Then, the decision would take such metrics when applying the ontology alignment.

Absolute and relative correspondences along with statistics and probability constitute only few of the other metrics the machine learning can rely on to enhance the device matching.

12.2.2 Device Composition

Our approach tackles the device heterogeneity and provides interoperability between two equivalent devices. Currently, ATOPAI allows to load and detect mappings between two ontologies. The actual standard devices

are mono-functions, however, new multi-function devices are currently arriving on the market, such as the multi-function printer scanner device. Thus, a multi-function device has to be probably matched with other several mono-function devices.

To support the multi-functions devices, an investigation must be carried out to check how such devices are specified. Whether they announce their selfs on the network as two logical devices supporting independent services or as one logical device providing several independent services. If the services are independent then the matching is still supported by ATOPAI. However, if the services are highly dependent one on another, then ATOPAI must be extended to support multiple ontology matching.

More over, DOXEN will also need to be extended to visit and generate code from multiple ontologies. Additionally, the code generation templates will be impacted if the several equivalent mono-function devices support different plug and play protocol. Thus, the templates need to be extended to support several Base drivers API instead of currently only two base drivers API.

12.2.3 Security & Privacy

Plug and Play devices can be controlled on the network by any protocol compatible control point. Even worse, any control point can connect to an end user's media library and retrieve content saved on its Network Area Storage (NAS) device without going through any authentication or identity verification. Therefore, new discussions are initiated in order to define privacy roles on some devices. Meaning, that according to the end-user's identity distinct access authorizations are allowed. For instance, the end-user's owner will have a full access on its device content and use while a visitor will only use a part of the provided services of the device.

Thus, according to the authentication method, an adaptation must be carried out on the generated proxies. Indeed, the proxies should be able to provide to the applications and end user's authentication mechanisms similar to those used on the real device. More over, if necessary, the generated proxy must establish secured channels between the application and the real device.

12.2.4 Adaptation Code

The following perspective aims to improve the actual version of ATOPAI. In fact, currently an expert can add through ATOPAI an adaptation code using the high level *adaptation* and the *external service* APIs. We made the assumption that the code is correct, thus, it is directly injected in the templates without any verification. Therefore, a parser like the *JavaCC* can be added to parse and verify the code before attaching it to the ontology.

Additionally, the expert specifies using ATOPAI the imported packages needed in his adaptation code. So far, ATOPAI does not allow to add external dependencies from specific Jars and other native libraries. Extending ATOPAI to support adding external packages dependencies would be handy for some adaptation use cases.

12.2.5 DOXEN

In the current global architecture, DOXEN is placed in the home network on a Set-Top-Box. It generates on the fly specific proxies based on the ontology alignments. Another alternative is still to be considered. It consists in placing DOXEN on the operator's site and download the generated proxies on demand to the digital homes. Placing DOXEN on the operator's site could accelerate the proxy generation time since more powerful machines can be used. A comparison of the two alternatives need to be considered.

"Patience is bitter, but it's fruit is sweet."

– Aristotle

Appendix A

Publications

- *INSIGHT: Interoperability and Service Management for the Digital Home.* Charbel El Kaed, Loïc Petit, Maxime Louvel, Antonin Chazalet, Yves Denneulin and François-Gaël Ottogalli. **Middleware 2011 (Industrial Track)**, ACM/IFIP/USENIX 12th International Middleware Conference, December 2011, Lisbon, Portugal. Acceptance Rate: 33%.

- *Dynamic Service Adaptation for Plug and Play Device Interoperability.* Charbel El Kaed, Yves Denneulin and François-Gaël Ottogalli. **CNSM 2011**, 7th International Conference on Network and Service Management, IEEE. October 2011, Paris, France. Acceptance Rate: 15%, (24/161).

- *On the Fly Proxy Generation for Home Devices Interoperability.* Charbel El Kaed, Yves Denneulin and François-Gaël Ottogalli. **MDM 2011 (Industrial Track)**, 12th IEEE International Conference on Mobile Data Management, Sweden, 2011. Acceptance Rate [Main Track]: 25%, (22/88).

- *Génération de mandataire pour l'interopérabilité des services.* Charbel El Kaed, Yves Denneulin et François-Gaël Ottogalli. **CFSE 2011**, 8ème Conférence Française des Systèmes d'Exploitation, St Malo, France, 2011.

- *Combining Ontology Alignment with Model Driven Engineering Techniques for Home Devices Interoperability.* Charbel El Kaed, Yves Denneulin, François-Gaël Ottogalli, and Luis Felipe Melo Mora. **SEUS 2010**, 8th IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, LNCS, October 13-15, 2010, Waidhofen/Ybbs, Austria.

- *CBay: Extending auction algorithm for component placement in the Internet of machines.* Charbel El Kaed, Yves Denneulin and François-Gaël Ottogalli. **Percom 2010 Workshops**, eighth Annual IEEE International Conference on Pervasive Computing and Communications, IEEE PerWare2010. 29 March- 2 April 2010, Mannheim, Germany.

- *CBay: enchères pour le redéploiement de composants sur l'internet des machines.* Charbel El Kaed, Yves Denneulin and François-Gaël Ottogalli. **Ubimob'09**, Proceedings of the 5th French-Speaking Conference on Mobility and Ubiquity Computing, ACM. July, 2009, Lille, France.

Appendix B

Additional Examples and Figures

B.1 An OWL Ontology Example

Listing B.1 shows the ontology of Figure 5.3 expressed using the OWL language detailed in section 5.1.3.1.

```
<rdf:RDF xmlns="http://www.d-ontologie.fr/myPhD/context/home/fig24.owl#"
 xml:base="http://www.d-ontologie.fr/myPhD/context/home/fig24.owl"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
 xmlns:xsp="http://www.owl-ontologies.com/2005/08/07/xsp.owl#"
 xmlns:swrl="http://www.w3.org/2003/11/swrl#"
 xmlns:protege="http://protege.stanford.edu/plugins/owl/protege#"
 xmlns:swrlb="http://www.w3.org/2003/11/swrlb#"
 xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:owl="http://www.w3.org/2002/07/owl#">
<owl:Ontology rdf:about=""/>
 <owl:Class rdf:ID="Brightness">
  <rdfs:subClassOf rdf:resource="#PhysicalProperty"/>
 </owl:Class>
 <owl:ObjectProperty rdf:ID="characterizes">
  <owl:inverseOf rdf:resource="#isCharacterizedBy"/>
 </owl:ObjectProperty>
 <owl:Class rdf:ID="ClimStation">
  <rdfs:subClassOf rdf:resource="#TempServ"/>
 </owl:Class>
 <owl:Class rdf:ID="Device"/>
 <owl:Class rdf:ID="Distraction_device">
  <rdfs:subClassOf rdf:resource="#Device"/>
 </owl:Class>
 <owl:Class rdf:ID="Game">
  <rdfs:subClassOf rdf:resource="#Distraction_device"/>
 </owl:Class>
 <owl:Class rdf:ID="HotelRoom">
  <rdfs:subClassOf rdf:resource="#Room"/>
 </owl:Class>
 <HotelRoom rdf:ID="1345"/>
 <owl:Class rdf:ID="IndoorLocation">
   <rdfs:subClassOf rdf:resource="#Location"/>
  </owl:Class>
 <owl:ObjectProperty rdf:ID="informs">
  <rdfs:domain rdf:resource="#Sensor"/>
  <rdfs:range rdf:resource="#PhysicalProperty"/>
```

```
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:ID="isCharacterizedBy">
   <rdfs:domain rdf:resource="#Location"/>
   <owl:inverseOf rdf:resource="#characterizes"/>
   <rdfs:range rdf:resource="#PhysicalProperty"/>
  </owl:ObjectProperty>
  <owl:Class rdf:ID="jsd678-mk2">
   <rdfs:subClassOf rdf:resource="#ClimStation"/>
  </owl:Class>
  <jsd678-mk2 rdf:ID="jsd67898200"/>
  <owl:ObjectProperty rdf:ID="locatedIn">
  <rdfs:domain rdf:resource="#Device"/>
  <rdfs:range rdf:resource="#Location"/>
  </owl:ObjectProperty>
  <owl:Class rdf:ID="Location"/>
  <owl:Class rdf:ID="Moisture">
   <rdfs:subClassOf rdf:resource="#PhysicalProperty"/>
  </owl:Class>
  <owl:Class rdf:ID="PhysicalProperty"/>
  <owl:Class rdf:ID="Room">
   <rdfs:subClassOf rdf:resource="#IndoorLocation"/>
  </owl:Class>
  <owl:Class rdf:ID="Sensor">
   <rdfs:subClassOf rdf:resource="#Device"/>
  </owl:Class>
  <owl:Class rdf:ID="Temperature">
   <rdfs:subClassOf rdf:resource="#PhysicalProperty"/>
  </owl:Class>
  <owl:Class rdf:ID="TempServ">
   <rdfs:subClassOf rdf:resource="#Sensor"/>
  </owl:Class>
  <owl:Class rdf:ID="TVSet">
   <rdfs:subClassOf rdf:resource="#Distraction_device"/>
  </owl:Class>
 </rdf:RDF>
```

Listing B.1: OWL Description [Pierson 09] of the Ontology in Figure 5.3

B.2 A DPWS PrintTicket Element

Figure B.1 shows a PrintTicket element structure. The PrintTicket element is used as an input by the CreatePrintJob DPWS operation.

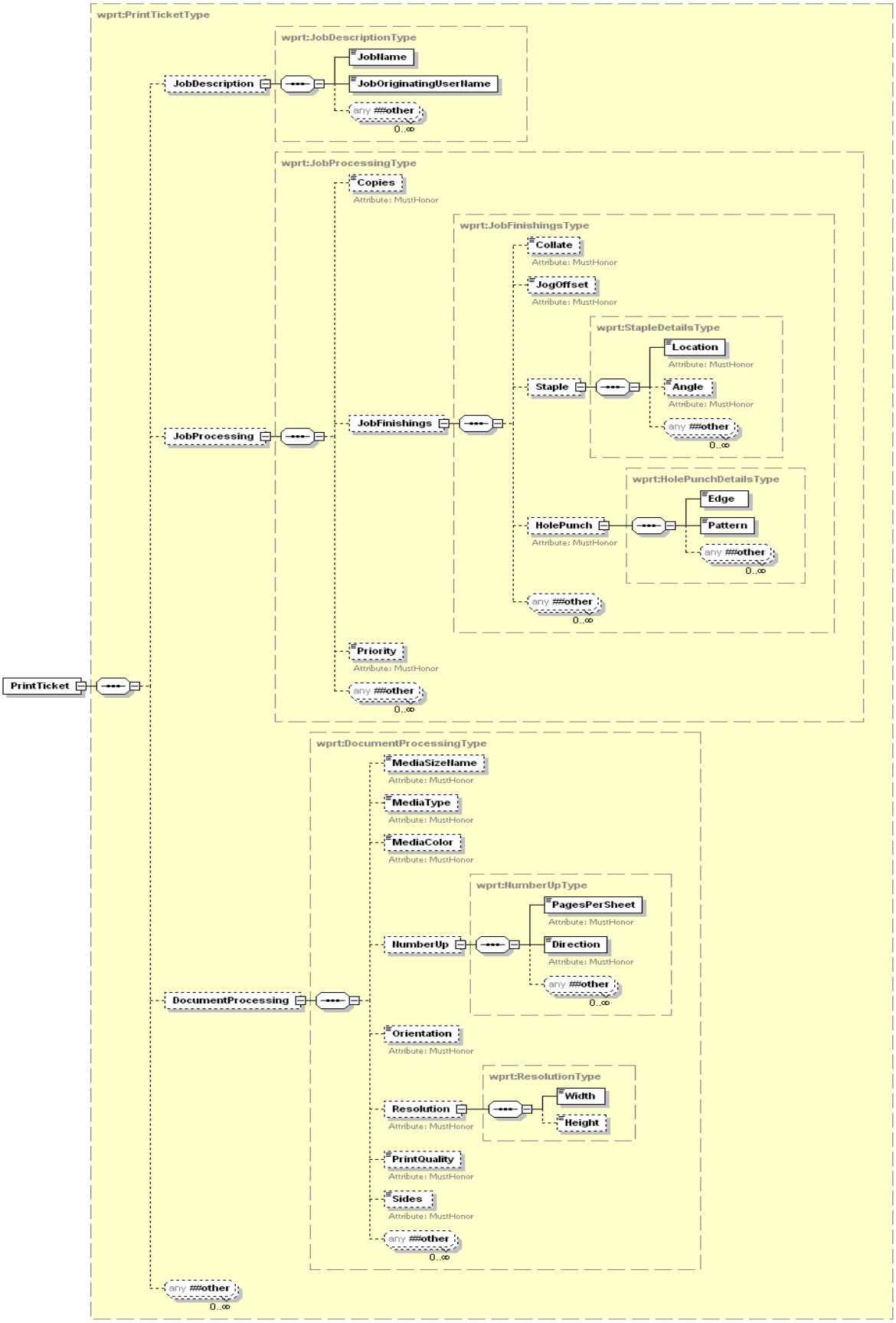


Figure B.1: A Standard DPWS Print Ticket Element [Microsoft 07]

B.3 DPWS Ontology Generation Example

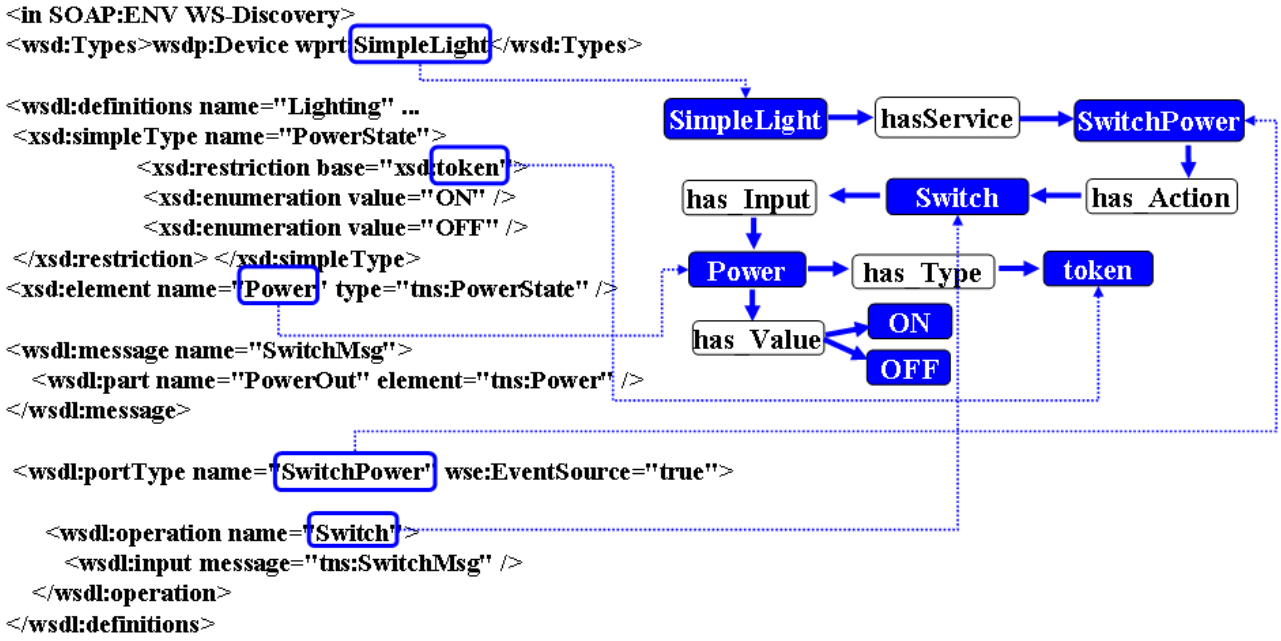


Figure B.2: DPWS Ontology Generation from a WSDL description

B.4 UPnP Binary Light Generated Ontology in OWL

```
<?xml version="1.0"?>
<rdf:RDF xmlns="http://OWLWriter/UPnPWriter.owl#" xml:base="http://OWLWriter/UPnPWriter.owl"
      xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
      xmlns:owl="http://www.w3.org/2002/07/owl#"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
      xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
      xmlns:UPnPWriter="http://OWLWriter/UPnPWriter.owl#">
    <owl:Ontology rdf:about="http://OWLWriter/UPnPWriter.owl"/>
    <!-- ///// ObjectProperties /////-->
    <owl:ObjectProperty rdf:about="http://OWLWriter/UPnPWriter.owl#UPnP_is_instance_of_type"/>
    <owl:ObjectProperty rdf:about="http://OWLWriter/UPnPWriter.owl#has_UPnP_Action"/>
    <owl:ObjectProperty rdf:about="http://OWLWriter/UPnPWriter.owl#has_UPnP_Input"/>
    <owl:ObjectProperty rdf:about="http://OWLWriter/UPnPWriter.owl#has_UPnP_Instance_Input"/>
    <owl:ObjectProperty rdf:about="http://OWLWriter/UPnPWriter.owl#has_UPnP_Instance_Output"/>
    <owl:ObjectProperty rdf:about="http://OWLWriter/UPnPWriter.owl#has_UPnP_Output"/>
    <owl:ObjectProperty rdf:about="http://OWLWriter/UPnPWriter.owl#has_UPnP_Service"/>
    <!-- ///// Data properties ///// -->
  <owl:DatatypeProperty rdf:about="http://OWLWriter/UPnPWriter.owl#
      Status_has_UPnP_StateVariable_Type"/>
  <owl:DatatypeProperty rdf:about="http://OWLWriter/UPnPWriter.owl#
      Target_has_UPnP_StateVariable_Type"/>
  <owl:DatatypeProperty rdf:about="http://OWLWriter/UPnPWriter.owl#has_UPnP_Device_Type"/>
  <owl:DatatypeProperty rdf:about="http://OWLWriter/UPnPWriter.owl#has_UPnP_Service_ID"/>
  <owl:DatatypeProperty rdf:about="http://OWLWriter/UPnPWriter.owl#has_UPnP_Service_Type"/>
  <owl:DatatypeProperty rdf:about="http://OWLWriter/UPnPWriter.owl#has_UPnP_Service_Version"/>
    <!--///// Classes ///// -->
```

```
<owl:Class rdf:about="http://OWLWriter/UPnPWriter.owl#BinaryLight">
    <rdfs:subClassOf rdf:resource="http://OWLWriter/UPnPWriter.owl#UPnP_Device"/>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="http://OWLWriter/UPnPWriter.owl#
                has_UPnP_Device_Type"/>
            <owl:hasValue>urn:schemas-upnp-org:device:BinaryLight:1</owl:hasValue>
        </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="http://OWLWriter/UPnPWriter.owl#has_UPnP_Service"/>
        <owl:someValuesFrom rdf:resource="http://OWLWriter/UPnPWriter.owl#SwitchPower"/>
      </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:about="http://OWLWriter/UPnPWriter.owl#GetStatus">
    <rdfs:subClassOf rdf:resource="http://OWLWriter/UPnPWriter.owl#UPnP_Action"/>
    <rdfs:subClassOf>
      <owl:Restriction>
         <owl:onProperty rdf:resource="http://OWLWriter/UPnPWriter.owl#
             has_UPnP_Instance_Output"/>
         <owl:someValuesFrom rdf:resource="http://OWLWriter/UPnPWriter.owl#ResultStatus"/>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
          <owl:onProperty rdf:resource="http://OWLWriter/UPnPWriter.owl#has_UPnP_Output"/>
          <owl:someValuesFrom rdf:resource="http://OWLWriter/UPnPWriter.owl#Status"/>
      </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:about="http://OWLWriter/UPnPWriter.owl#GetTarget">
    <rdfs:subClassOf rdf:resource="http://OWLWriter/UPnPWriter.owl#UPnP_Action"/>
    <rdfs:subClassOf>
       <owl:Restriction>
          <owl:onProperty rdf:resource="http://OWLWriter/UPnPWriter.owl#has_UPnP_Output"/>
          <owl:someValuesFrom rdf:resource="http://OWLWriter/UPnPWriter.owl#Target"/>
       </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="http://OWLWriter/UPnPWriter.owl#
                has_UPnP_Instance_Output"/>
            <owl:someValuesFrom rdf:resource="http://OWLWriter/UPnPWriter.owl#
                RetTargetValue"/>
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:about="http://OWLWriter/UPnPWriter.owl#NewTargetValue">
    <rdfs:subClassOf rdf:resource="http://OWLWriter/UPnPWriter.owl#Target"/>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="http://OWLWriter/UPnPWriter.owl#
                UPnP_is_instance_of_type"/>
            <owl:someValuesFrom rdf:resource="http://OWLWriter/UPnPWriter.owl#Target"/>
        </owl:Restriction>
```

```
        </rdfs:subClassOf>
    </owl:Class>
    <owl:Class rdf:about="http://OWLWriter/UPnPWriter.owl#ResultStatus">
        <rdfs:subClassOf rdf:resource="http://OWLWriter/UPnPWriter.owl#Status"/>
        <rdfs:subClassOf>
            <owl:Restriction>
                <owl:onProperty rdf:resource="http://OWLWriter/UPnPWriter.owl#
                    UPnP_is_instance_of_type"/>
                <owl:someValuesFrom rdf:resource="http://OWLWriter/UPnPWriter.owl#Status"/>
            </owl:Restriction>
        </rdfs:subClassOf>
    </owl:Class>
    <owl:Class rdf:about="http://OWLWriter/UPnPWriter.owl#RetTargetValue">
        <rdfs:subClassOf rdf:resource="http://OWLWriter/UPnPWriter.owl#Target"/>
        <rdfs:subClassOf>
            <owl:Restriction>
                <owl:onProperty rdf:resource="http://OWLWriter/UPnPWriter.owl#
                    UPnP_is_instance_of_type"/>
                <owl:someValuesFrom rdf:resource="http://OWLWriter/UPnPWriter.owl#Target"/>
            </owl:Restriction>
        </rdfs:subClassOf>
    </owl:Class>
    <owl:Class rdf:about="http://OWLWriter/UPnPWriter.owl#SetTarget">
        <rdfs:subClassOf rdf:resource="http://OWLWriter/UPnPWriter.owl#UPnP_Action"/>
        <rdfs:subClassOf>
            <owl:Restriction>
                <owl:onProperty rdf:resource="http://OWLWriter/UPnPWriter.owl#
                    has_UPnP_Instance_Input"/>
                <owl:someValuesFrom rdf:resource="http://OWLWriter/UPnPWriter.owl#
                    NewTargetValue"/>
            </owl:Restriction>
        </rdfs:subClassOf>
        <rdfs:subClassOf>
          <owl:Restriction>
             <owl:onProperty rdf:resource="http://OWLWriter/UPnPWriter.owl#has_UPnP_Input"/>
             <owl:someValuesFrom rdf:resource="http://OWLWriter/UPnPWriter.owl#Target"/>
          </owl:Restriction>
        </rdfs:subClassOf>
    </owl:Class>
    <owl:Class rdf:about="http://OWLWriter/UPnPWriter.owl#Status">
        <rdfs:subClassOf rdf:resource="http://OWLWriter/UPnPWriter.owl#UPnP_StateVariable"/>
        <rdfs:subClassOf>
            <owl:Class>
                <owl:oneOf rdf:parseType="Collection">
                    <rdf:Description rdf:about="http://OWLWriter/UPnPWriter.owl#true"/>
                    <rdf:Description rdf:about="http://OWLWriter/UPnPWriter.owl#false"/>
                </owl:oneOf>
            </owl:Class>
        </rdfs:subClassOf>
        <rdfs:subClassOf>
            <owl:Restriction>
                <owl:onProperty rdf:resource="http://OWLWriter/UPnPWriter.owl#
                    Status_has_UPnP_StateVariable_Type"/>
                <owl:someValuesFrom rdf:resource="http://www.w3.org/2001/XMLSchema#boolean"/>
            </owl:Restriction>
        </rdfs:subClassOf>
    </owl:Class>
```

```
<owl:Class rdf:about="http://OWLWriter/UPnPWriter.owl#SwitchPower">
    <rdfs:subClassOf rdf:resource="http://OWLWriter/UPnPWriter.owl#UPnP_Service"/>
    <rdfs:subClassOf>
      <owl:Restriction>
         <owl:onProperty rdf:resource="http://OWLWriter/UPnPWriter.owl#has_UPnP_Action"/>
         <owl:someValuesFrom rdf:resource="http://OWLWriter/UPnPWriter.owl#GetStatus"/>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="http://OWLWriter/UPnPWriter.owl#
                has_UPnP_Service_ID"/>
            <owl:hasValue>urn:upnp-org:serviceId:SwitchPower:1</owl:hasValue>
        </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="http://OWLWriter/UPnPWriter.owl#
                has_UPnP_Service_Version"/>
            <owl:hasValue>1</owl:hasValue>
        </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="http://OWLWriter/UPnPWriter.owl#
                has_UPnP_Service_Type"/>
            <owl:hasValue>urn:schemas-upnp-org:service:SwitchPower:1</owl:hasValue>
        </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
         <owl:onProperty rdf:resource="http://OWLWriter/UPnPWriter.owl#has_UPnP_Action"/>
         <owl:someValuesFrom rdf:resource="http://OWLWriter/UPnPWriter.owl#GetTarget"/>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="http://OWLWriter/UPnPWriter.owl#has_UPnP_Action"
                />
            <owl:someValuesFrom rdf:resource="http://OWLWriter/UPnPWriter.owl#SetTarget"/>
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:about="http://OWLWriter/UPnPWriter.owl#Target">
    <rdfs:subClassOf rdf:resource="http://OWLWriter/UPnPWriter.owl#UPnP_StateVariable"/>
    <rdfs:subClassOf>
        <owl:Class>
            <owl:oneOf rdf:parseType="Collection">
                <rdf:Description rdf:about="http://OWLWriter/UPnPWriter.owl#true"/>
                <rdf:Description rdf:about="http://OWLWriter/UPnPWriter.owl#false"/>
            </owl:oneOf>
        </owl:Class>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="http://OWLWriter/UPnPWriter.owl#
                Target_has_UPnP_StateVariable_Type"/>
```

```
                <owl:someValuesFrom rdf:resource="http://www.w3.org/2001/XMLSchema#boolean"/>
            </owl:Restriction>
        </rdfs:subClassOf>
    </owl:Class>
    <owl:Class rdf:about="http://OWLWriter/UPnPWriter.owl#UPnP_Action">
        <rdfs:subClassOf>
          <owl:Restriction>
            <owl:onProperty rdf:resource="http://OWLWriter/UPnPWriter.owl#has_UPnP_Input"/>
            <owl:someValuesFrom rdf:resource="http://OWLWriter/UPnPWriter.owl#
                UPnP_StateVariable"/>
          </owl:Restriction>
        </rdfs:subClassOf>
        <rdfs:subClassOf>
          <owl:Restriction>
              <owl:onProperty rdf:resource="http://OWLWriter/UPnPWriter.owl#has_UPnP_Output"/>
              <owl:someValuesFrom rdf:resource="http://OWLWriter/UPnPWriter.owl#
                  UPnP_StateVariable"/>
          </owl:Restriction>
        </rdfs:subClassOf>
    </owl:Class>
    <owl:Class rdf:about="http://OWLWriter/UPnPWriter.owl#UPnP_Device">
        <rdfs:subClassOf>
           <owl:Restriction>
             <owl:onProperty rdf:resource="http://OWLWriter/UPnPWriter.owl#has_UPnP_Service"/>
             <owl:someValuesFrom rdf:resource="http://OWLWriter/UPnPWriter.owl#UPnP_Service"/>
           </owl:Restriction>
        </rdfs:subClassOf>
    </owl:Class>
    <owl:Class rdf:about="http://OWLWriter/UPnPWriter.owl#UPnP_Service">
        <rdfs:subClassOf>
          <owl:Restriction>
            <owl:onProperty rdf:resource="http://OWLWriter/UPnPWriter.owl#has_UPnP_Action"/>
            <owl:someValuesFrom rdf:resource="http://OWLWriter/UPnPWriter.owl#UPnP_Action"/>
          </owl:Restriction>
        </rdfs:subClassOf>
    </owl:Class>
    <owl:Class rdf:about="http://OWLWriter/UPnPWriter.owl#UPnP_StateVariable"/>
    <!-- ///// Individuals /////-->
    <owl:NamedIndividual rdf:about="http://OWLWriter/UPnPWriter.owl#false">
        <rdf:type rdf:resource="http://OWLWriter/UPnPWriter.owl#Status"/>
        <rdf:type rdf:resource="http://OWLWriter/UPnPWriter.owl#Target"/>
    </owl:NamedIndividual>
    <owl:NamedIndividual rdf:about="http://OWLWriter/UPnPWriter.owl#true">
        <rdf:type rdf:resource="http://OWLWriter/UPnPWriter.owl#Status"/>
        <rdf:type rdf:resource="http://OWLWriter/UPnPWriter.owl#Target"/>
    </owl:NamedIndividual>
</rdf:RDF>
<!-- Generated by the OWL API (version 3.0.0.1413) http://owlapi.sourceforge.net -->
```

Listing B.2: A UPnP BinrayLight Generated Ontology

B.5 An Alignment Result between a UPnP and DPWS Lights in Align format


```
<?xml version='1.0' encoding='utf-8' standalone='no'?>
<rdf:RDF xmlns='http://knowledgeweb.semanticweb.org/heterogeneity/alignment'
         xmlns:ns0='http://knowledgeweb.semanticweb.org/heterogeneity/alignment'
         xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
         xmlns:xsd='http://www.w3.org/2001/XMLSchema#'
         xmlns:align='http://knowledgeweb.semanticweb.org/heterogeneity/alignment#'>
<Alignment>
  <xml>yes</xml>
  <level>0</level>
  <type>**</type>
  <ns0:time>1560</ns0:time>
  <ns0:method>com.francetelecom.align.smoaStrategy</ns0:method>
  <onto1>
    <Ontology rdf:about="http://OWLWriter/UPnPWriter.owl">
      <location>file:/C:/homedevices/UPnPWriter_BinaryLight.owl</location>
      <formalism>
        <Formalism align:name="OWL2.0" align:uri="http://www.w3.org/2002/07/owl#"/>
      </formalism>
    </Ontology>
  </onto1>
  <onto2>
    <Ontology rdf:about="http://OWLWriter/DPWSWriter.owl">
      <location>file:/C:/homedevices/DPWSWriter_SimpleLight.owl</location>
      <formalism>
        <Formalism align:name="OWL2.0" align:uri="http://www.w3.org/2002/07/owl#"/>
      </formalism>
    </Ontology>
  </onto2>
  <map>
    <Cell>
      <entity1 rdf:resource='http://OWLWriter/UPnPWriter.owl#Target'/>
      <entity2 rdf:resource='http://OWLWriter/DPWSWriter.owl#LightLevelTarget'/>
      <relation>=</relation>
      <measure rdf:datatype='http://www.w3.org/2001/XMLSchema#float'>0.66</measure>
    </Cell>
  </map>
  <map>
    <Cell>
      <entity1 rdf:resource='http://OWLWriter/UPnPWriter.owl#SetLoadLevelTarget'/>
      <entity2 rdf:resource='http://OWLWriter/DPWSWriter.owl#SetLevel'/>
      <relation>=</relation>
      <measure rdf:datatype='http://www.w3.org/2001/XMLSchema#float'>0.71</measure>
    </Cell>
  </map>
  <map>
    <Cell>
      <entity1 rdf:resource='http://OWLWriter/UPnPWriter.owl#GetMinLevel'/>
      <entity2 rdf:resource='http://OWLWriter/DPWSWriter.owl#GetLevel'/>
      <relation>=</relation>
      <measure rdf:datatype='http://www.w3.org/2001/XMLSchema#float'>0.83</measure>
    </Cell>
  </map>
  <map>
```

```
      <Cell>
        <entity1 rdf:resource='http://OWLWriter/UPnPWriter.owl#GetStatus'/>
        <entity2 rdf:resource='http://OWLWriter/DPWSWriter.owl#GetStatus'/>
        <relation>=</relation>
        <measure rdf:datatype='http://www.w3.org/2001/XMLSchema#float'>1.0</measure>
      </Cell>
    </map>
    <map>
      <Cell>
        <entity1 rdf:resource='http://OWLWriter/UPnPWriter.owl#DimmingService'/>
        <entity2 rdf:resource='http://OWLWriter/DPWSWriter.owl#Dimming'/>
        <relation>=</relation>
        <measure rdf:datatype='http://www.w3.org/2001/XMLSchema#float'>0.76</measure>
      </Cell>
    </map>
    <map>
      <Cell>
        <entity1 rdf:resource='http://OWLWriter/UPnPWriter.owl#BinaryLight'/>
        <entity2 rdf:resource='http://OWLWriter/DPWSWriter.owl#SimpleLight'/>
        <relation>=</relation>
        <measure rdf:datatype='http://www.w3.org/2001/XMLSchema#float'>0.69</measure>
      </Cell>
    </map>
    <map>
      <Cell>
        <entity1 rdf:resource='http://OWLWriter/UPnPWriter.owl#SwitchPower'/>
        <entity2 rdf:resource='http://OWLWriter/DPWSWriter.owl#SwitchPower'/>
        <relation>=</relation>
        <measure rdf:datatype='http://www.w3.org/2001/XMLSchema#float'>1.0</measure>
      </Cell>
    </map>
    <map>
      <Cell>
        <entity1 rdf:resource='http://OWLWriter/UPnPWriter.owl#LoadLevelTarget'/>
        <entity2 rdf:resource='http://OWLWriter/DPWSWriter.owl#LightLevelTarget'/>
        <relation>=</relation>
        <measure rdf:datatype='http://www.w3.org/2001/XMLSchema#float'>0.67</measure>
      </Cell>
    </map>
    <map>
      <Cell>
        <entity1 rdf:resource='http://OWLWriter/UPnPWriter.owl#GetLoadLevelStatus'/>
        <entity2 rdf:resource='http://OWLWriter/DPWSWriter.owl#GetLevel'/>
        <relation>=</relation>
        <measure rdf:datatype='http://www.w3.org/2001/XMLSchema#float'>0.75</measure>
      </Cell>
    </map>
  </Alignment>
  </rdf:RDF>
```

Listing B.3: An Alignment Result between a UPnP and DPWS Lights in the Align format

B.6 Screen Shots of the UPnP-Android Based Home Controller

In the left part of Figure B.3, the Android UPnP Home Controller is showed where the intel UPnP Light and the UPnP-DPWS proxy are detected. The right part shows the basic interaction interface which allows to retrieve the actual state of a UPnP light and turn it on or off.

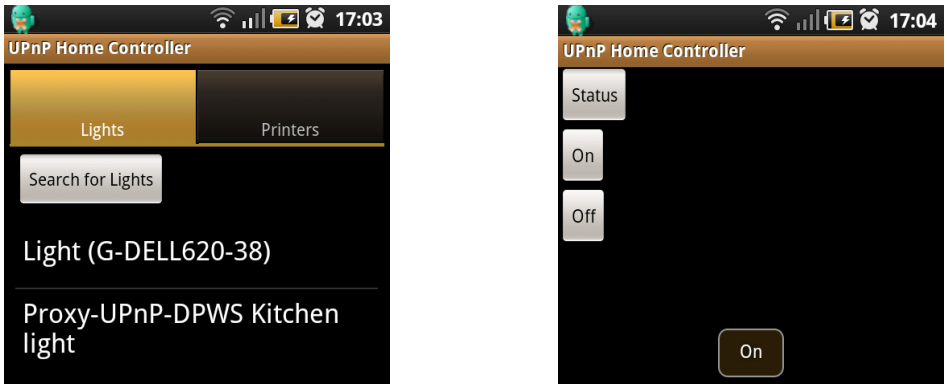


Figure B.3: UPnP Lights detected by the UPnP-Android based Home Controller Application

Figure B.4: UPnP Printer detected by the UPnP-Android based Home Controller Application

In the left part of Figure B.4, the Android UPnP Home Controller is showed where the UPnP-DPWS Proxy Printer is detected. The right part shows the basic interaction interface which allows to print a document from a url. The other parameters such as the number of pages, color, etc are hidden but set by default. The application is only a proof of concept and will be enhanced later.

Appendix C

Detailed Alignment Results

Table C.1: Legend

Symbol	Explanation
–	Expert Added
$\triangleleft$	Default value set using ATOPAI
x	No equivalence, default value is used
$\approx$	Detected but need adaptation

C.1 SMOA Printers Alignment Results

Table C.2: SMOA Mapping between a DPWS and a UPnP Printer without Similarity Propagation

UPnP DPWS Printer devices			SMOA	
Type	UPnP	DPWS	t=0.63	t=0.25
Device	Printer	PrinterDevice	0.89	0.89
Service	PrintEnhanced	PrintService	-	0.62
Action	CancelJob	CancelJob	1	1
In SV	JobId	CancelJobRequest/JobId	1	1
Action	CreateURIJob	Sequential(CreatePrintJob, SendDocument)	0.85/–	0.85/–
	CreateURIJob	Sequential(CreatePrintJob, AddDocument)	0.85/–	0.85/–
In SV	Copies	CreatePrintJobRequest/PrintTicket/JobProcessing/Copies	1	1
	JobName	CreatePrintJobRequest/PrintTicket/JobDescription/JobName	1	1
	JobOriginating-UserName	CreatePrintJobRequest/PrintTicket/JobDescription/-JobOriginatingUserName	1	1
	NumberUp	SendDocumentRequest/DocumentProcessing-/NumberUp/Direction	-	-
	PrintQuality	SendDocumentRequest/DocumentProcessing/PrintQuality	1	1
	Sides	SendDocumentRequest/DocumentProcessing/Sides	1	1
	Orientation-Requested	SendDocumentRequest/DocumentProcessing/Orientation	0.91	0.91
	MediaType	SendDocumentRequest/DocumentProcessing/MediaType	1	1
	MediaSize	SendDocumentRequest/DocumentProcessing/MediaSizeName	–	–
	Document-Format	SendDocumentRequest/DocumentDescription/Format	–	–
	SourceURI	AddDocumentRequest/DocumentURL	–	–
	–	SendDocumentRequest/DocumentDescription/Compression ◁ None		
	–	SendDocumentRequest/DocumentData (Fetch)	x	x
	–	SendDocumentRequest/LastDocument ◁ true	x	x
Out SV	JobId	CreatePrintJobResponse/JobId	1	1

Table C.3: SMOA Mapping between a DPWS and a UPnP Printer (Continuation of Table C.2)

UPnP DPWS Printer devices			SMOA	
Type	UPnP	DPWS	t=0.63	t=0.25
Action	GetJob-Attributes	GetJobElements	–	–
In SV	JobId	GetJobElementsRequest/JobId	1	1
	–	GetJobElementsRequest/RequestedElements/Name ◁ tns:JobStatus,tns:PrintTicket	x	x
Out SV	JobMediaSheets-Completed	GetJobElementsResponse/JobElements/ElementData/-JobStatus/MediaSheetsCompleted	0.97	0.97
	JobName	GetJobElementsResponse/JobElements/ElementData/-PrintTicket/JobDescription/JobName	1	1
	JobOriginating-UserName	GetJobElementsResponse/JobElements/ElementData/-PrintTicket/JobDescription/JobOriginatingUserName	1	1
Action	GetPrinter-Attributes	Union(GetActiveJobs, GetPrinterElements)	–/0.72	–/0.72
In SV	–	GetPrinterElementsRequest/-RequestedElements/Name ◁ tns:PrinterStatus	x	x
Out SV	JobId	GetActiveJobsResponse/-ActiveJobs/JobSummary/JobId	1	1
	PrinterState	GetPrinterElementsResponse/PrinterElements/-ElementData/PrinterStatus/PrinterState	1	1
	PrinterState-Reasons	GetPrinterElementsResponse/-PrinterElements/ElementData/-PrinterStatus/PrinterStateReasons/PrinterStateReason	0.99	0.99
	JobIdList	To be supported though ActiveJobs - return multiple Jobs to be put into the List,	≈	≈

Table C.4: SMOA False Mapping between a DPWS and a UPnP Printer(Continuation of Table C.3)

False Matching: SMOA			t=0.63	t=0.25
Action	GetJob-Attributes	GetJobHistory		0.64
	GetMargins	GetActiveJobs		0.46
	GetMediaList	GetJobHistory	0.67	0.67
SV	NumberUp	GetActiveJobsResponse/ActiveJobs/JobsSummary/-NumberOfDocuments	1	1
	MediaSize	GetPrinterElementResponse/PrinterElements/-ElementData/PrinterConfiguration/-InputBins/InputBinEntry/MediaSize	1	1
	Document-Format	AddDocumentRequest/DocumentDescription/DocumentId	0.85	0.85
	ARG_TYPE-CriticalAttributes	GetPrinterElementsResponse/PrinterElements/-.../Consumables/ConsumableEntry/Type		0.63
	InternetConnect-State	GetPrinterElementsResponse/PrinterElements/...-/PrinterStatus/PrinterState	0.76	0.76
	PageMargins	AddDocumentRequest/DocumentProcessing/NumberUp/-PagesPerSheet		0.57
	FullBleed-Supported	GetPrinterElementsResponse/.../ColorSupported	0.68	0.68
	DataSink	SendDocumentRequest/DocumentData		0.55

Summary: SMOA	t=0.63	t=0.25
Success	20/28	21/28
Percentage	71%	75%
False Match	6	11

C.2 SMOA++ Printers Alignment Results

Table C.5: SMOA++ Mapping between a DPWS and a UPnP Printer without Similarity Propagation

UPnP DPWS Light devices			SMOA++	
Type	UPnP	DPWS	t=0.63	t=0.25
Device	Printer	PrinterDevice	0.67	0.67
Service	PrintEnhanced	PrintService	-	0.5
Action	CancelJob	CancelJob	1	1
In SV	JobId	CancelJobRequest/JobId	1	1
Action	CreateURIJob	Sequential(CreatePrintJob, SendDocument)	0.67/–	0.67/–
	CreateURIJob	Sequential(CreatePrintJob, AddDocument)	0.67/–	0.67/–
In SV	Copies	CreatePrintJobRequest/PrintTicket/JobProcessing/Copies	1	1
	JobName	CreatePrintJobRequest/PrintTicket/JobDescription/JobName	1	1
	JobOriginating-UserName	CreatePrintJobRequest/PrintTicket/JobDescription/-JobOriginatingUserName	1	1
	NumberUp	SendDocumentRequest/DocumentProcessing/-NumberUp/Direction	-	-
	PrintQuality	SendDocumentRequest/DocumentProcessing/PrintQuality	1	1
	Sides	SendDocumentRequest/DocumentProcessing/Sides	1	1
	Orientation-Requested	SendDocumentRequest/DocumentProcessing/Orientation	0.67	0.67
	MediaType	SendDocumentRequest/DocumentProcessing/MediaType	1	1
	MediaSize	SendDocumentRequest/DocumentProcessing/MediaSizeName	–	–
	Document-Format	SendDocumentRequest/DocumentDescription/Format	0.67	0.67
	SourceURI	AddDocumentRequest/DocumentURL	–	–
	–	SendDocumentRequest/DocumentDescription/Compression ◁ None		
	–	SendDocumentRequest/DocumentData (Fetch)	x	x
	–	SendDocumentRequest/LastDocument ◁ true	x	x
Out SV	JobId	CreatePrintJobResponse/JobId	1	1

Table C.6: SMOA++ Mapping between a DPWS and a UPnP Printer (Continuation of Table C.5)

UPnP DPWS Light devices			SMOA++	
Type	UPnP	DPWS	t=0.63	t=0.25
Action	GetJob-Attributes	GetJobElements	–	–
In SV	JobId	GetJobElementsRequest/JobId	1	1
	–	GetJobElementsRequest/RequestedElements/Name ◁ tns:JobStatus,tns:PrintTicket	x	x
Out SV	JobMediaSheets-Completed	GetJobElementsResponse/JobElements/ElementData/-JobStatus/MediaSheetsCompleted	0.89	0.89
	JobName	GetJobElementsResponse/JobElements/ElementData/-PrintTicket/JobDescription/JobName	1	1
	JobOriginating-UserName	GetJobElementsResponse/JobElements/ElementData/-PrintTicket/JobDescription/JobOriginatingUserName	1	1
Action	GetPrinter-Attributes	Union(GetActiveJobs, GetPrinterElements)	–/–	–/0.57
In SV	–	GetPrinterElementsRequest/-RequestedElements/Name ◁ tns:PrinterStatus	x	x
Out SV	JobId	GetActiveJobsResponse/-ActiveJobs/JobSummary/JobId	1	1
	PrinterState	GetPrinterElementsResponse/PrinterElements/-ElementData/PrinterStatus/PrinterState	1	1
	PrinterState-Reasons	GetPrinterElementsResponse/-PrinterElements/ElementData/-PrinterStatus/PrinterStateReasons/PrinterStateReason	0.67	0.67
	JobIdList	To be supported through ActiveJobs - return multiple Jobs to be put into the List,	≈	≈

Table C.7: SMOA++ False Mapping between a DPWS and a UPnP Printer (Continuation of Table C.6)

False Matching: SMOA++			t=0.63	t=0.25
Action	GetJob-Attributes	GetActiveJobs	0.75	0.75
	GetMargins	GetJobElements	–	0.29
	GetMediaList	GetJobElements	–	0.29
SV	NumberUp	GetActiveJobsResponse/ActiveJobs/JobsSummary/-NumberOfDocuments	–	0.42
	MediaSize	GetPrinterElementResponse/PrinterElements/-ElementData/PrinterConfiguration/-InputBins/InputBinEntry/MediaSize	1	1
	InternetConnect-State	GetActiveJobsResponse/ActiveJobs/JobSummary/JobState	–	0.4
	FullBleed-Supported	GetPrinterElementsResponse/.../CollationSupported	–	0.4
	DataSink	SendDocumentRequest/DocumentData	–	0.5

Summary: SMOA++	t=0.63	t=0.25
Success	20/28	22/28
Percentage	71%	78%
False Match	2	8

C.3 SMOA++ Printers Alignment Results with a Similarity Propagation

Table C.8: SMOA++ Mapping between a DPWS and a UPnP Printer with a Similarity Propagation

UPnP DPWS Light devices			SMOA++			
Type	UPnP	DPWS	t_P			
			1	0.8	0.4	$\overline{t_P}$
Device	Printer	PrinterDevice	0.67	0.67	0.67	0.67
Service	PrintEnhanced	PrintService	0.75	0.7	0.67	0.5
Action	CancelJob	CancelJob	1	1	1	1
Action	CreateURIJob	Sequential(CreatePrintJob, SendDocument)	0.83/–	0.83/–	0.83/–	0.67/–
Action	GetJob-Attributes	GetJobElements	–	–	–	–
Action	GetPrinter-Attributes	Union(GetActiveJobs, GetPrinterElements)	–/0.78	–/0.78	–/0.78	–/0.5
False Matching: SMOA++			1	0.8	0.4	$\overline{t_P}$
Action	GetJob-Attributes	GetActiveJobs	0.75	0.81	0.81	0.75
	GetMargins	GetJobElements	0.29	0.29	0.29	0.29
	GetMediaList	GetJobElements	0.29	0.29	0.29	0.29
Summary: SMOA++			1	0.8	0.4	$\overline{t_P}$
Success (t=0.25)			22/28	22/28	22/28	22/28
Percentage (t=0.25)			78%	78%	78%	78%
False Match (t=0.25)			8	8	8	8
Success (t=0.63)			22/28	22/28	22/28	20/28
Percentage (t=0.63)			78%	78%	78 %	71%
False Match (t=0.63)			2	2	2	2

Bibliography

[Aipperspach 08] Ryan Aipperspach, Ben Hooker & Allison Woodruff. *The heterogeneous home*. In Proceedings of the 10th international conference on Ubiquitous computing, UbiComp '08, pages 222–231, New York, NY, USA, 2008. ACM.

[Akkiraju 05] Rama Akkiraju, Joel Farrell, John Miller, Meenakshi Nagarajan, Marc-Thomas Schmidt, Amit Sheth & Kunal Verma. *Web Service Semantics - WSDL-S*. http://www.w3.org/Submission/WSDL-S/, 2005.

[Alves 07] Alexandre Alves, Assaf Arkin, Sid Askary, Charlton Barreto, Ben Bloch, Francisco Curbera & et al. *Web Services Business Process Execution Language Version 2.0*. docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html, 2007.

[Apache a] Apache. http://felix.apache.org/site/index.html.

[Apache b] Felix Apache. *UPnP Base Driver*. http://felix.apache.org/site/apache-felix-upnp.html.

[Apple 05] Apple. *Bonjour Printing Specification*, 2005.

[Apple 11a] Apple. *DNS-Based Service Discovery*. Internet Engineering Task Force, Internet-Draft, February 2011.

[Apple 11b] Apple. *Multicast DNS*. Internet Engineering Task Force , Internet-Draft, 2011.

[Aumueller 05] David Aumueller, Hong-Hai Do, Sabine Massmann & Erhard Rahm. *Schema and ontology matching with COMA++*. In SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data, pages 906–908, New York, NY, USA, 2005. ACM.

[Battle 05] Steve Battle, Abraham Bernstein, Harold Boley, Benjamin Grosof, Michael Gruninger & et al. *Semantic Web Services Framework (SWSF)*. http://www.w3.org/Submission/2005/07/, 2005.

[Bechhofer 04] Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter Patel-Schneider & Andrea Stein. *Web Ontology Language*. http://www.w3.org/TR/owl-features/, 2004.

[Bézivin 05] Jean Bézivin. *On the unification power of models*. Software and System Modeling, vol. 4, pages 171–188, 2005.

[Bigelow 99] Stephen J. Bigelow. The plug and play book. McGraw-Hill (New York), 1999.

[Bluetooth] Bluetooth. www.bluetooth.com.

[Bonino 08a] D. Bonino, E. Castellina & F. Corno. *The DOG Gateway: Enabling Ontology-based Intelligent Domotic Environments.* IEEE TRANSACTIONS ON CONSUMER ELECTRONICS., vol. vol. 54/4 ISSN: 0098-3063,, pages pp. 1656–1664, 2008.

[Bonino 08b] Dario Bonino, Emiliano Castellina & Fulvio Corno. *Uniform Access to Domotic Environments through Semantics.* 2008.

[Bonino 09] Dario Bonino & Fulvio Corno. *Interoperation Modeling for Intelligent Domotic Environments.* In AmI '09: Proceedings of the European Conference on Ambient Intelligence, pages 143–152, Berlin, Heidelberg, 2009. Springer-Verlag.

[Bonino 10] Dario Bonino & Fulvio Corno. *Rule-based intelligence for domotic environments.* Automation in Construction, vol. 19, no. 2, pages 183 – 196, 2010.

[Bonjour] Bonjour. http://developer.apple.com/networking/bonjour/specs.html.

[Bottaro 06] André Bottaro & Anne Gérodolle. *Comments on ISO/IEC WD 14543-5-1: Intelligent Grouping and Resource Sharing (IGRS) for HES Class 2 and Class 3 - Part 5-1: Core Protocol*, 2006.

[Bottaro 07a] André Bottaro. *RFP 86 - DPWS Discovery Base Driver.* pagesperso-orange.fr/andre.../rfp-86-DPWSDiscoveryBaseDriver.pdf, 2007.

[Bottaro 07b] André Bottaro, Anne Gérodolle & Philippe Lalanda. *Pervasive Service Composition in the Home Network.* In AINA '07: Proceedings of the 21nd International Conference on Advanced Information Networking and Applications, pages 378–385, Washington, DC, USA, 2007. IEEE Computer Society.

[Bottaro 08a] André Bottaro. *Home SOA : composition contextuelle de services dans les reseauxd'equipements pervasifs.* PhD thesis, Université Joseph Fourier, Grenoble I, 2008.

[Bottaro 08b] André Bottaro & Anne Gérodolle. *Home SOA - facing protocol heterogeneity in pervasive applications.* In Proceedings of the 5th international conference on Pervasive services: ICPS'08, pages 73–80, New York, NY, USA, 2008. ACM.

[Broadband] Forum Broadband. *Industry Adoption of TR-069 Specifications Grows to Address Multimedia Device Management.* DSL Forum TR-069 Interoperability Plugfest Event, http://www.broadband-forum.org/.

[Broadband 10a] Forum Broadband. *MR-230: TR-069 Deployment Scenarios Issue: 1.* http://www.broadband-forum.org/marketing/download/mktgdocs/MR-230.pdf, August 2010.

[Broadband 10b] Forum Broadband. *Proposed Draft-174: Remote Management of non TR-069 Devices.* http://www.broadband-forum.org/, February 2010.

[Budanitsky 06] Alexander Budanitsky & Graeme Hirst. *Evaluating WordNet-based Measures of Lexical Semantic Relatedness.* Comput. Linguist., vol. 32, pages 13–47, March 2006.

[Cetina 07] Carlos Cetina, Estefania Serral, Javier Munoz & Vicente Pelechano. *Tool Support for Model Driven Development of Pervasive Systems.* In MOMPES '07: Proceedings of the Fourth International Workshop on Model-Based Methodologies for Pervasive and Embedded Software, pages 33–44, Washington, DC, USA, 2007. IEEE Computer Society.

[Chazalet 11] Antonin Chazalet, Sebastien Bolle & Serge Martin. *Analyzing the Digital Homes Quality of Service.* The 4th Int' Workshop on Service Science and Systems (COMPSAC), 2011.

[Chen 05] H. Chen, T. Finin & A. Joshi. The soupa ontology for pervasive computing. Springer, 2005.

[Chen 09] Chaoand Chen & Abdelsalam Helal. *Device Integration in SODA Using the Device Description Language.* In Ninth Annual International Symposium on Applications and the Internet, SAINT 2009, Bellevue, Washington, USA, July 20-24, 2009, Proceedings, pages 100–106. IEEE Computer Society, 2009.

[Christensen 01] Erik Christensen, Francisco Curbera, Greg Meredith & Sanjiva Weerawarana. *Web Service Definition Language (WSDL).* Rapport technique, 2001.

[Clocksin 03] W.F. Clocksin & C.S. Mellish. Programming in prolog. Springer-Verlag, 2003.

[Cohen 98] J. Cohen & S. Aggarwal. *General Event Notification Architecture Base.* INTERNET DRAFT, July 1998.

[Combs 02] Garron Combs. *Plug and Play, or Plug and Pray.* CS 350: Computer Organization Spring Section 2, 2002.

[Coopman 10] T. Coopman, W. Theetaert, D. Preuveneers & Y. Berbers. *A user-oriented and context-aware service orchestration framework for dynamic home automation systems.* In Ambient Intelligence and Future Trends - International Symposium on Ambient Intelligence. Springer, 2010.

[DAA 05] *Unofficial DAAP protocol documentation.* http://tapjam.net/daap/, April 12, 2005.

[David 10] Jérôme David, Jérôme Euzenat, Franois Scharffe & Cassia Trojahn dos Santos. *The Alignment API 4.0.* Semantic Web Journal, 2010.

[Davis 93] Randall Davis, Howard Shrobe & Peter Szolowits. *What is a Knowledge Representation?* AI Magazine, 14(1):17-33, 1993, 1993.

[Devedžić 02] Vladan Devedžić. *Understanding ontological engineering.* Communication ACM, vol. 45, pages 136–144, April 2002.

[Dixon 10] Colin Dixon, Ratul Mahajan, Sharad Agarwal, A. J. Brush, Bongshin Lee, Stefan Saroiu & Victor Bahl. *The home needs an operating system (and an app store).* In Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks, Hotnets '10, pages 18:1–18:6, New York, NY, USA, 2010. ACM.

[DLNA 03] DLNA. *Digital Living Network Alliance.* www.dlna.org, 2003.

[Do 02] Hong-Hai Do & Erhard Rahm. *COMA: a system for flexible combination of schema matching approaches.* In Proceedings of the 28th international conference on Very Large Data Bases, VLDB '02, pages 610–621. VLDB Endowment, 2002.

[Droms 97] R. Droms. *Dynamic Host Configuration Protocol.* RFC 2131, 1997.

[DTMF] DTMF. *Web Services for Management.* http://www.dmtf.org/.

[Edwards 01] W. Keith Edwards & Rebecca E. Grinter. *At Home with Ubiquitous Computing: Seven Challenges.* In Proceedings of the 3rd international conference on Ubiquitous Computing, UbiComp '01, pages 256–272, London, UK, UK, 2001. Springer-Verlag.

[El Kaed 10] Charbel El Kaed, Yves Denneulin, François-Gaël Ottogalli & Luis Felipe Melo Mora. *Combining ontology alignment with model driven engineering techniques for home devices interoperablity.* In Proceedings of the 8th IFIP WG 10.2 international conference on Software technologies for embedded and ubiquitous systems, SEUS'10, pages 71–82, Berlin, Heidelberg, 2010. Springer-Verlag.

[El Kaed 11a] Charbel El Kaed, Yves Denneulin & François-Gaël Ottogalli. *Dynamic Service Adaptation for Plug and Play Device Interoperability.* In 7th International Conference on Network and Service Management (CNSM 2011), Paris, France, October 2011.

[El Kaed 11b] Charbel El Kaed, Yves Denneulin & François-Gaël Ottogalli. *On the Fly Proxy Generation for Home Devices Interoperability,.* 12th International Conference on Mobile Data Management, Lulea, Sweden, June 2011.

[El Kaed 11c] Charbel El Kaed, Loïc Petit, Maxime Louvel, Antoine Chazalet, Yves Denneulin & François-Gäel Ottogalli. *INSIGHT: Interoperability and Service Management for the Digital Home.* In Middleware 2011 (Industrial Track), ACM/IFIP/USENIX 12th International Middleware Conference,, Lisbon, Portugal, December 2011.

[Elbyed 09] Abdeltif Elbyed. *ROMIE, une approche d'alignement d'ontologies à base d'instances.* PhD thesis, Institut National Des Telecommunications, 2009.

[Euzenat 04] Jérôme Euzenat. *An API for Ontology Alignment.* In Sheila McIlraith, Dimitris Plexousakis & Frank van Harmelen, editeurs, The Semantic Web ISWC 2004, volume 3298 of *Lecture Notes in Computer Science*, pages 698–712. Springer Berlin / Heidelberg, 2004.

[Euzenat 07] Jérôme Euzenat & Pavel Shvaiko. Ontology matching. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[Fellbaum 98] Christiane Fellbaum. *WordNet: An Electronic Lexical Database.* Cambridge, MA: MIT Press, 1998.

[Frank 66] Norman Frank & et. Structural models: An introduction to the theory of directed graphs. John WileySons, 1966.

[Frénot 10] Stéphane Frénot, Noha Ibrahim, Frédéric Le Mouël, Amira Ben Hamida, Julien Ponge, Mathieu Chantrel & Denis Beras. *ROCS: a Remotely Provisioned OSGi Framework for Ambient Systems.* In Network Operations and Management Symposium, pages 503–510, Osaka, Japan, April 2010. IEEE/IFIP.

[Fujii 05] Keita Fujii & Tatsuya Suda. *Semantics-based dynamic service composition,.* IEEE Journal on Selected Areas in Communications (JSAC), special issue on Autonomic Communication Systems, vol. 23, pages 2361–2372, 2005.

[Gašević 06] Dragan Gašević, Dragan Djuric, Vladan Devedzic & Bran Selic. Model driven architecture and ontology development. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[Giunchiglia 06] Fausto Giunchiglia, Fiona McNeill & Mikalai Yatskevich. *Web Service Composition via Semantic Matching of Interaction Specifications.* Rapport technique, University of Trento, Italy, 2006.

[Goland 99] Yaron Goland, Ting Cai, Paul Leach & Ye Gu. *Simple Service Discovery Protocol.* Internet Engineering Task Force, INTERNET DRAFT, October 1999.

[Gonzalez 78] R.C. Gonzalez & M.G Thomas. Syntactic pattern recognition: an introduction. Addison Wesley Publishing Company, Reading,MA, 1978.

[Hagget 67] P. Hagget & R.J. Chorley. *Models, paradigms and new geography.* In Models in Geography, Methuen, London, 1967.

[Hall 10] R. Hall, K. Pauls, S. McCulloch & D. Savage. Osgi in action: Creating modular applications in java. Manning Pubs Co Series. Manning Publications, 2010.

[Hamming 50] Richard Hamming. *Error detecting and error correcting codes.* Rapport technique, Bell System Technical Journal, 1950.

[Harrington 02] D. Harrington, R. Presuhn & B. Wijnen. *An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks.* RFC 3411, December 2002.

[Helal 05] Sumi Helal, William Mann, Hicham El-Zabadani, Jeffrey King, Youssef Kaddoura & Erwin Jansen. *The Gator Tech Smart House: a programmable pervasive space*, 2005.

[Helal 09] Sumi Helal & Chao Chen. *The Gator Tech Smart House: Enabling Technologies and Lessons Learned.* In i-CREATe '09: Proceedings of the 3rd International Convention on Rehabilitation Engineering & Assistive Technology, 2009.

[Helaoui 11] Rim Helaoui, Mathias Niepert & Heiner Stuckenschmidt. *Recognizing Interleaved and Concurrent Activities: A Statistical-Relational Approach.* PerCom, 2011.

[Horrocks 03] Ian Horrocks, Jurgen Angele, Stefan Decker, Michael Kifer, Benjamin Grosof & Gerd Wagner. *Where Are the Rules?* IEEE Intelligent Systems, vol. 18, pages 76–83, September 2003.

[Horrocks 04] Ian Horrocks, Peter Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof & Mike Dean. *SWRL: A Semantic Web Rule Language.* www.w3.org/Submission/SWRL, 2004.

[Ibrahim 08] Noha Ibrahim. *Spontaneous Integration of Services in Pervasive Environments.* PhD thesis, Institut National des Sciences Appliques de Lyon, 2008.

[IETF 07] IETF. *6LowPAN: Transmission of IPv6 Packets over IEEE 802.15.4 Networks.* RFC 4944, 2007.

[IGRS] IGRS. *Intelligent Grouping and Resource Sharing.* http://www.igrs.org/.

[IGRS 06] IGRS. *Information technology – Home electronic system (HES) architecture – Part 5-1: Intelligent grouping and resource sharing for Class 2 and Class 3 – Core protocol.* www.iso.org, 2006.

[IGRS 07] IGRS. *Information technology - 4 Home Electronic System (HES) Architecture 5 Intelligent Grouping and Resource Sharing for HES 6 Class 2 and Class 3 Part 6: Service type.* www.iso.org, 2007.

[INRIA] INRIA. *Align API.* http://alignapi.gforge.inria.fr.

[Jain 10] Atishay Jain & Ashish Tanwer. *Modified Epc Global Network Architecture of Internet of Things for High Load Rfid Systems.* In Proceedings of International Conference on Advances in Computer Science, 2010.

[Jouault 08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin & Ivan Kurtev. *ATL: A model transformation tool.* Sci. Comput. Program., vol. 72, pages 31–39, June 2008.

[Kalfoglou 01] Yannis Kalfoglou. Exploring ontologies, volume 1, page 863887. World Scientific Publishing, 2001.

[Kindberg 02] Tim Kindberg & Armando Fox. *System Software for Ubiquitous Computing.* IEEE Pervasive Computing, vol. 1, pages 70–81, January 2002.

[Konnex 04] Konnex. *KNX Handbook v. 1,1.* Rapport technique, Konnex Association:, 2004.

[Kurkovsky 07] S. Kurkovsky. *Pervasive computing: Past, present and future.* In Information and Communications Technology, 2007. ICICT 2007. ITI 5th International Conference on, pages 65 –71, dec. 2007.

[Levenshtein 65] Vladimir Levenshtein. *Binary codes capable of correcting deletions, insertions, and reversals. In Russian. English Translation in Soviet Physics Doklady, 10(8) p. 707710, 1966.* Rapport technique, Doklady akademii nauk SSSR, 1965.

[Lupton 07] William Lupton, John Blackford, Mike Digdon, Tim Spets, Greg Bathrick & Heather Kirksey. *TR-069 CPE WAN Management Protocol v1.1.* Rapport technique, The Broadband Forum., 2007.

[Lyytinen 02] Kalle Lyytinen & Youngjin Yoo. *Issues and Challenges in Ubiquitous Computing: Introduction.* Commun. ACM, vol. 45, pages 62–65, December 2002.

[MacKenzie 06] M. MacKenzie, K. Laskey, F. McCabe, P Brown & R. Metz. *Reference model for service oriented architecture 1.0.* Rapport technique, OASIS, 2006.

[Martin 04] David Martin, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne, Evren Sirin, Naveen Srinivasan & Katia Sycara. *Semantic Markup for Web Services.* http://www.w3.org/Submission/OWL-S/, 2004.

[McIlraith 03] Sheila A. McIlraith & David L. Martin. *Bringing Semantics to Web Services.* IEEE Intelligent Systems, vol. 18, pages 90–93, January 2003.

[Melnik 02] Sergey Melnik, Hector Garcia-Molina & Erhard Rahm. *Similarity Flooding: A Versatile Graph Matching Algorithm and Its Application to Schema Matching.* In Proceedings of the 18th International Conference on Data Engineering, ICDE '02, pages 117–, Washington, DC, USA, 2002. IEEE Computer Society.

[Michelson 06] Brenda Michelson. *Event-Driven Architecture Overview.* Rapport technique, Patricia Seybold Group, 2006.

[Microsoft 06] Microsoft. *Web Services On Device, A windows Rally white paper.* White paper, Microsoft, 2006.

[Microsoft 07] Microsoft. *Standard DPWS Printer specifications.* http://msdn.microsoft.com/en-us/windows/hardware/gg463146.aspx, 2007.

[Miori 06] Vittorio Miori, L. Tarrini, M. Manca & G. Tolomei. *An open standard solution for domotic interoperability.* IEEE Transactions on Consumer Electronics, vol. 52, no. 1, pages 97–103, février 2006.

[Miori 10] Vittorio Miori, Dario Russo & Massimo Aliberti. *Domotic technologies incompatibility becomes user transparent.* Communications of the ACM, vol. 53, no. 1, pages 153–157, 2010.

[Mokhtar 06] Sonia Mokhtar, Anupam Kaul, Nikolaos Georgantas & Valérie Issarny. *Efficient semantic service discovery in pervasive computing environments.* In Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware, Middleware '06, pages 240–259, New York, NY, USA, 2006. Springer-Verlag New York, Inc.

[Mokhtar 07] Sonia Mokhtar. *Semantic Middleware for Service-Oriented Pervasive Computing.* PhD thesis, University of Paris 6, 2007.

[Mokhtar 08] Sonia Mokhtar, Davy Preuveneers, Nikolaos Georgantas, Valérie Issarny & Yolande Berbers. *EASY: Efficient semAntic Service discoverY in pervasive computing environments with QoS and context support.* J. Syst. Softw., vol. 81, pages 785–808, May 2008.

[Moon 05] Kyeong-deok Moon, Young-hee Lee, Chang-eun Lee & Young-sung Son. *Design of a universal middleware bridge for device interoperability in heterogeneous home network middleware.* In IEEE Transactions on Consumer Electronics, volume 51, pages 314–318, février 2005.

[Moore 65] Gordon Moore. *Cramming more components onto integrated circuits.* Electronics, Volume 38, Number 8, April 1965.

[Mora 10] Luis Felipe Melo Mora. Interopérabilité des modèles de données. Master's thesis, Université Joseph Fourier, Grenoble, France, 2010.

[Munoz 04] Javier Munoz, Pelechano, Vicente & J. Fons. *Model Driven Development of Pervasive Systems.* In International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES), pages 3–14, 2004.

[Nain 08] Grégory Nain, Erwan Daubert, Olivier Barais & Jean-Marc Jézéquel. *Using MDE to Build a Schizophrenic Middleware for Home/Building Automation.* In ServiceWave '08: Proceedings of the 1st European Conference on Towards a Service-Based Internet, pages 49–61, Berlin, Heidelberg, 2008. Springer-Verlag.

[Nakazawa] J. Nakazawa. *uMiddle.* http://www.ht.sfc.keio.ac.jp/ jin/research/uMiddle/.

[Nakazawa 06] Jin Nakazawa, W. Keith Edwards, Hideyuki Tokuda & Umakishore Ramachandran. *A Bridging Framework for Universal Interoperability in Pervasive Systems.* In in Pervasive Systems, ICDCS 2006. IEEE Computer Society, 2006.

[Nikolaidis 07] A.E. Nikolaidis, S. Papastefanos, G.A. Doumenis, G.I. Stassinopoulos & M.P.K. Drakos. *Local and remote management integration for flexible service provisioning to the home.* Communications Magazine, IEEE, vol. 45, no. 10, pages 130–138, October 2007.

[Noy 00] Natalya Noy & Mark A. Musen. *PROMPT: Algorithm and Tool for Automated Ontology Merging and Alignment.* In Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence, pages 450–455. AAAI Press, 2000.

[Noy 01] Natalya Noy & Deborah McGuinness. *Ontology Development 101: A Guide to Creating Your First Ontology.* Rapport technique KSL-01-05, Knowledge Systems, AI Laboratory, Stanford University, 2001.

[Noy 04] Natalya Noy. *Semantic Integration: A Survey Of Ontology-Based Approaches.* SIGMOD Record, vol. 33, page 2004, 2004.

[OASIS 04] OASIS. *Introduction to UDDI: Important Features and Functional Concepts.* http://uddi.org/pubs/uddi-tech-wp.pdf, 2004.

[OASIS 09a] OASIS. *Devices Profile for Web Services Version 1.1.* http://docs.oasis-open.org/ws-dd/dpws/1.1/os/wsdd-dpws-1.1-spec-os.html, 2009.

[OASIS 09b] OASIS. *Web Services Dynamic Discovery (WS-Discovery) Version 1.1.* http://docs.oasis-open.org/ws-dd/discovery/1.1/os/wsdd-discovery-1.1-spec-os.pdf, July 2009.

[OMA] OMA. *Open Mobile Alliance, Device Management,.* http://www.openmobilealliance.org/.

[OMG 97] Object-Management-Group OMG. *UML Specification version 1.1.* www.omg.org/cgi-bin/doc?ad/97-08-11, 1997.

[OMG 03] Object-Management-Group OMG. *MDA Guide Version 1.0.1.*, 2003.

[OMG 06] Object-Management-Group OMG. *Meta Object Facility Specifications.* www.omg.org/spec/MOF/2.0/, 2006.

[OSGi] Alliance OSGi. www.osgi.org.

[OSGi 09a] OSGi. *OSGi Service Platform Core Specification, Release 4, Version 4.2,.* http://www.osgi.org, June 2009.

[OSGi 09b] OSGi. *OSGi Service Platform Service Compendium, Release 4, Version 4.2,.* http://www.osgi.org, August 2009.

[OWL] *The OWL API.* `http://owlapi.sourceforge.net/`.

[Paolucci 02] Massimo Paolucci, Takahiro Kawamura, Terry R. Payne & Katia P. Sycara. *Semantic Matching of Web Services Capabilities.* In ISWC '02: Proceedings of the First International Semantic Web Conference on The Semantic Web, pages 333–347, London, UK, 2002. Springer-Verlag.

[Parra 09] Jorge Parra, M. Anwar Hossain, Aitor Uribarren, Eduardo Jacob & Abdulmotaleb El Saddik. *Flexible Smart Home Architecture using Device Profile for Web Services: a Peer-to-Peer Approach.* International Journal of Smart Home, vol. vol 3, no 2, 2009.

[Petit 11] Loïc Petit, Claudi Roncancio, Cyril Labbé & François-Gäel Ottogalli. *DomVision : Intergiciel de gestion de données pour l'environnement domestique.* 27èmes journées Bases de Données Avancées, Rabat, Maroc, 2011.

[Pierson 09] Jerome Pierson. *Une infrastructure de gestion de l'information de contexte pour l'intelligence ambiante.* PhD thesis, Université Joseph Fourier - Grenoble 1, 2009.

[Prud'hommeaux 04] Eric Prud'hommeaux & Andy Seaborne. *SPARQL Query Language for RDF.* www.w3.org/TR/rdfsparql-query, 2004.

[Quan 08] LI Quan, SHU Yuanzhong, TAO Chenggong & WANG Lingsheng. *Device Common Resource Description and Management with CIM in Industry.* International Conference on Computer and Electrical Engineering, 2008.

[Redondo 08] Rebeca P. Diaz Redondo & et al. *Enhancing Residential Gateways: A Semantic OSGi Platform.* IEEE Intelligent Systems, 2008.

[Robertson 06] Dave Robertson, Fausto Giunchiglia, Frank van Harmelen, Maurizio Marchese, Marta Sabou, Marco Schorlemmer, Nigel Shadbolt, Ronnie Siebes, Carles Sierra, Chris Walton, Srinandan Dasmahapatra, Dave Dupplaw, Paul Lewis, Mikalai Yatskevich, Spyros Kotoulas, Adrian Perreau de Pinninck & Antonis Loizou. *Open Knowledge: Semantic Webs Through Peer - to - Peer Interaction.* Rapport technique, University of Trento, Italy, 2006.

[Roman 05] Dumitru Roman, Uwe Keller, Holger Lausen, Jos de Bruijn, Rubén Lara, Michael Stollberg, Axel Polleres, Cristina Feier, Christoph Bussler & Dieter Fensel. *Web Service Modeling Ontology.* Applied Ontology, vol. 1, no. 1, pages 77–106, 2005.

[Sabran 10] Thierry Sabran. *Development of a GUI for ontology alignment validation.* Grenoble INP, ENSIMAG, 2010. Internship report.

[Schmidt 06] D.C. Schmidt. *Guest Editor's Introduction: Model-Driven Engineering.* Computer, vol. 39, no. 2, pages 25 – 31, February 2006.

[Serral 08] Estefanía Serral, Pedro Valderas & Vicente Pelechano. *A Model Driven Development Method for Developing Context-Aware Pervasive Systems.* In UIC '08: Proceedings of the 5th international conference on Ubiquitous Intelligence and Computing, pages 662–676, Berlin, Heidelberg, 2008. Springer-Verlag.

[Sharma 98] R. Sharma, V.I. Pavlovic & T.S. Huang. *Toward multimodal human-computer interface.* Proceedings of the IEEE, vol. 86, no. 5, pages 853 –869, may 1998.

[Smarthome 04] Smarthome. *Powerlinc programming manual*, 2004.

[SOA4D a] SOA4D. *DPWS OSGi Base Driver.* http://forge.soa4d.org/projects/osgi-dpwsdriver/.

[SOA4D b] SOA4D. *Service Oriented Architecture 4 Devices.* https://forge.soa4d.org/.

[Spets 10] Tim Spets & Alex Fedosseev. *Common Application Layer.* Broadband Forum, Home Technical Working Group, 2010.

[Staab 09] Steffen Staab & Rudi Studer. Handbook on ontologies. Springer, 2nd edition, 2009.

[Steele 90] Guy L. Steele Jr. Common lisp: the language (2nd ed.). Digital Press, Newton, MA, USA, 1990.

[Steinberg 05] Daniel Steinberg & Stuart Cheshire. *Zero Configuration Networking: The Definitive Guide.* O'Reilly, 1st Edition, December 2005.

[Stoilos 05] Giorgos Stoilos, Giorgos Stamou & Stefanos Kollias. *A String Metric for Ontology Alignment.* International Semantic Web Conference, pages 624–637, 2005.

[Studer 07] R. Studer, S. Grimm & A. Abecker. Semantic web services: concepts, technologies, and applications. Springer eBooks collection: Computer science. Springer, 2007.

[Tigli 09a] Jean-Yves Tigli, Stéphane Lavirotte, Gaëtan Rey, Vincent Hourdin, Daniel Cheung-Foo-Wo, Eric Callegari & Michel Riveill. *WComp Middleware for Ubiquitous Computing: Aspects and Composite Event-based Web Services.* Annals of Telecommunications (AoT), vol. 64, no. 3-4, pages 197–214, April 2009.

[Tigli 09b] Jean-Yves Tigli, Stephane Lavirotte, Gaëtan Rey, Vincent Hourdin & Michel Riveill. *Lightweight Service Oriented Architecture for Pervasive Computing.* CoRR, vol. abs/1102.5193, 2009.

[UPnP] UPnP. *Universal Plug and Play.* http://www.upnp.org/.

[UPnP 02] UPnP. *UPnP Audio Video Architecture.* http://www.upnp.org/specs/av/UPnP-av-AVArchitecture-v1-20020622.pdf, June 2002.

[UPnP 03] Forum UPnP. *SwithcPower:1 Service Template Version 1.01.* http://www.upnp.org/standardizeddcps/lighting.asp, November 2003.

[UPnP 06a] UPnP. *Standard UPnP Printer.* http://www.upnp.org/standardizeddcps/default.asp, October 28 2006.

[UPnP 06b] Forum UPnP. *UPnP Technology - The Simple, Seamless Home Network.* white paper, December 2006.

[UPnP 08] Forum UPnP. *UPnP Device Architecture 1.1, Document Revision.* www.upnp.org, October 2008.

[UPnP 10a] Forum UPnP. *UPnP DM Basic Management:1 Service Template, for UPnP version 1.0.* http://upnp.org/specs/dm/UPnP-dm-BasicManagement-v1-Service.pdf., July 2010.

[UPnP 10b] Forum UPnP. *UPnP DM Configuration Management:1 Service Template, for UPnP version 1.0.* http://upnp.org/specs/dm/UPnP-dm-ConfigurationManagement-v1-Service.pdf., July 2010.

[UPnP 10c] Forum UPnP. *UPnP DM Software Management:1 Service Template, for UPnP version 1.0.* http://upnp.org/specs/dm/UPnP-dm-SoftwareManagement-v1-Service.pdf., July 2010.

[UPnP 11] Forum UPnP. *Upnp Device Management- Simplify The Administration Of Your Devices.* http://www.upnp.org/, April 2011.

[Uschold 04] Michael Uschold & Michael Gruninger. *Ontologies and semantics for seamless connectivity.* SIGMOD Rec., vol. 33, pages 58–64, December 2004.

[Vallée 05] Mathieu Vallée, Fano Ramparany & Laurent Vercouter. *Flexible Composition of Smart Device Services.* In Laurence Tianruo Yang, Jianhua Ma, Makoto Takizawa & Timothy K. Shih, editeurs, PSC, pages 165–171. CSREA Press, 2005.

[Šváb Zamazal 10] Ondřej Šváb Zamazal, Vojtěch Svátek & Luigi Iannone. *Pattern-based ontology transformation service exploiting OPPL and OWL-API.* In Proceedings of the 17th international conference on Knowledge engineering and management by the masses, EKAW'10, pages 105–119, Berlin, Heidelberg, 2010. Springer-Verlag.

[W3C 99] W3C. *Resource Description Framework.* http://www.w3.org/RDF/, 1999.

[W3C 00] W3C. *Simple Object Access Protocol.* http://www.w3.org/TR/2000/NOTE-SOAP-20000508/, May 2000.

[W3C 01] W3C. *W3C XML Schema Definition Language.* http://www.w3.org/TR/xmlschema11-1/, http://www.w3.org/TR/xmlschema11-2/, 2001.

[W3C 05] W3C. *SOAP Message Transmission Optimization Mechanism.* http://www.w3.org/TR/soap12-mtom/, 2005.

[W3C 06a] W3C. *Web Services Addressing 1.0 - Core.* http://www.w3.org/TR/ws-addr-core/, 2006.

[W3C 06b] W3C. *Web Services Eventing.* http://www.w3.org/Submission/WS-Eventing/, 2006.

[W3C 06c] W3C. *Web Services Transfer.* http://www.w3.org/Submission/WS-Transfer/, 2006.

[W3C 11] W3C. *Web Services Metadata Exchange.* http://www.w3.org/TR/ws-metadata-exchange/, 2011.

[Wache 02] Holger Wache, Ubbo Visser & Thorsten Scholz. *Ontology Construction - An Iterative and Dynamic Task.* In Proceedings of the Fifteenth International Florida Artificial Intelligence Research Society Conference, pages 445–449. AAAI Press, 2002.

[Weiser 91] Mark Weiser. *The computer for the 21st century.* Scientific American, September 1991.

[WS4D 10] WS4D. *uDPWS.* http://code.google.com/p/udpws/, August 2010.

[Xie 09] Yuefang Xie & Xiangqian Ding. *Design and implementation of interoperable module between UPnP and IGRS.* In Proceedings of the 3rd international conference on Intelligent information technology application, IITA'09, pages 561–564, Piscataway, NJ, USA, 2009. IEEE Press.

[Yim 07] Hyung-Jun Yim, Il-Jin Oh, Yun-Young Hwang, Kyu-Chul Lee, Kangchan Lee & Lee. *Design of DPWS Adaptor for Interoperability between Web Services and DPWS in Web Services on Universal Networks.* In Proceedings of the 2007 International Conference on Convergence Information Technology, ICCIT '07, pages 1032–1039, Washington, DC, USA, 2007. IEEE Computer Society.

[Yousuf 07] M.S. Yousuf & M. El-Shafei. *Power Line Communications: An Overview - Part I.* In Innovations in Information Technology, 2007. IIT '07. 4th International Conference on, pages 218 –222, nov. 2007.

[ZigBee 09] ZigBee. *Understanding ZigBee RF4CE.* www.zigbee.org, July 2009.