Hasselt University
and transnational University of Limburg

Development and Deployment of Interactive Pervasive Applications for Ambient Intelligent Environments

Dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science
at the transnational University of Limburg
to be defended by

Geert Vanderhulst

Promotor: prof. dr. Kris Luyten
Co-promotor: prof. dr. Karin Coninx

2007 – 2010
Acknowledgments

Fruitful research is influenced by so many things, not the least by the people you meet and with whom you share daily life at home and at the office. They all deserve a ‘thank you’ for their direct or indirect impact on this dissertation.

In particular, I would like to thank Kris Luyten for his advice in the course of my research activities. His informal way of guiding PhD students stimulates creativity and has always motivated me to come up with new ideas and (‘undercover’?) results. Furthermore, special thanks go to Karin Coninx for letting me explore my research interests and providing guidance whenever needed. With Kris and Karin as (co-)promotor, the journey towards a PhD was a memorable and fun trip through research I can advice to anyone.

I am also grateful to the entire EDM crew – with Eddy Flerackers as leading director – for providing me with the opportunity to start a PhD in the first place, visit various nice places and meet fellow researchers from all over the world. Over the last years, it was a pleasure to have colleagues around who made life at the office more fun and with whom I worked together in one way or the other: Jo Vermeulen, Jan Meskens, Mieke Haesen, Petr Aksenov, Nasim Mahmud, Lode Vanacken, Kristof Verpoorten, Chris Vandervelpen, Tim Clerckx, Yves Vandriessche, Alexandre Demeure, Jan Van den Bergh and Maarten Cardinaels. Special thanks also go to Daniel Schreiber and Wim Laurier for their collaboration on shared topics of interest. This was a truly enjoyable experience that gave rise to various new insights.

On a more personal note, I want to express my gratitude to my family and friends for their understanding (sometimes literally, ain’t it brother?), time spent together (there is no plan B, ain’t it Koen?), distraction at work (presence is underrated, ain’t it Peter?), challenging minds (extraordinary, ain’t it Karen?) and support in general. Finally, I would like to thank Stephanie Vermosen for the many joyful and musical moments we spent together, and for her taking such good care of me whilst writing this text.
Abstract

Bringing together heterogeneous computing devices and appliances gives rise to a pervasive environment where resources interact with each other, such as a mobile phone telling the car’s stereo to mute. Applications deployed in such an environment should be sufficiently dynamic to cope with new configurations. This goes beyond building context-aware applications that assume a fixed environment as there is no guarantee the configuration does not evolve, making the software developed for the initial situation deprecated. We present models and frameworks that support the development and deployment of pervasive applications that allow the environment configuration to change over time.

Computer-augmented resources also tend to become physically simpler to use (e.g. less buttons) but become more complex to handle in their digital dimension (e.g. overloaded user interfaces). As a consequence, the behaviour of the pervasive applications leveraging these resources gets even more complex to understand and configure. This demands for tools that help developers and end-users inspect and manipulate the current state of the pervasive computing environment. We present tools that can be used to observe applications at runtime, by means of the messages they exchange, the events they generate and the behaviour they define.

Moreover, efficient discovery of nearby devices and accessible services is one of the preconditions to obtain a usable pervasive environment. Typical user interfaces in these environments hide the heterogeneity of the environment for the end-user which often makes it hard to perceive the provided functionality. We present meta-user interfaces as a means for exploring and controlling the environment. Furthermore, we offer user-oriented views on the user’s environment based on pictures of this environment. We show how users can model, explore and finally interact with complex pervasive environments using migratable (photo-based) user interfaces.
<table>
<thead>
<tr>
<th>Acknowledgments</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>Contents</td>
<td>viii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>x</td>
</tr>
<tr>
<td>List of Listings</td>
<td>xi</td>
</tr>
</tbody>
</table>

## 1 Introduction

1.1 Background .............................................. 1
1.2 Problem statement .................................... 2
  1.2.1 Merging real and digital worlds ................. 2
  1.2.2 Merging user and developer roles ............... 3
1.3 Towards do-it-yourself pervasive applications ...... 4

## I Modelling Pervasive Applications

2 Semantic Environment Model ............................. 9
2.1 Introduction .......................................... 9
2.2 Ontologies ............................................. 11
  2.2.1 Web Ontology Language ............................ 12
  2.2.2 Upper and domain ontologies ................... 14
2.3 Environment ontology ................................. 14
  2.3.1 Resources ........................................ 14
  2.3.2 Users and devices ................................ 16
  2.3.3 Services .......................................... 17
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3.4</td>
<td>Tasks and user interfaces</td>
<td>19</td>
</tr>
<tr>
<td>2.3.5</td>
<td>Groundings</td>
<td>21</td>
</tr>
<tr>
<td>2.3.6</td>
<td>Domain concepts</td>
<td>23</td>
</tr>
<tr>
<td>2.4</td>
<td>Distributed context management</td>
<td>24</td>
</tr>
<tr>
<td>2.4.1</td>
<td>Context on demand: aggregators</td>
<td>25</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Context on change: sensors</td>
<td>28</td>
</tr>
<tr>
<td>2.5</td>
<td>Discussion</td>
<td>29</td>
</tr>
<tr>
<td>3</td>
<td>Modelling Reactive Systems</td>
<td>31</td>
</tr>
<tr>
<td>3.1</td>
<td>Introduction</td>
<td>31</td>
</tr>
<tr>
<td>3.2</td>
<td>Behaviour rules</td>
<td>33</td>
</tr>
<tr>
<td>3.3</td>
<td>Behaviour ontology</td>
<td>33</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Events, conditions and actions</td>
<td>34</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Groundings</td>
<td>35</td>
</tr>
<tr>
<td>3.4</td>
<td>Distributed rule management</td>
<td>36</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Defining behaviour</td>
<td>36</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Explaining, correcting and predicting behaviour</td>
<td>38</td>
</tr>
<tr>
<td>3.5</td>
<td>Discussion</td>
<td>39</td>
</tr>
<tr>
<td>II</td>
<td>Developing and Deploying Pervasive Applications</td>
<td>41</td>
</tr>
<tr>
<td>4</td>
<td>Making Resources Talk</td>
<td>43</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>43</td>
</tr>
<tr>
<td>4.2</td>
<td>W2P framework</td>
<td>45</td>
</tr>
<tr>
<td>4.3</td>
<td>Messages: a vehicle for data exchange</td>
<td>47</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Addressing peers and groups</td>
<td>48</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Sending and receiving messages</td>
<td>49</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Monitoring the message flow</td>
<td>52</td>
</tr>
<tr>
<td>4.4</td>
<td>Case study: distributed media player</td>
<td>53</td>
</tr>
<tr>
<td>4.5</td>
<td>Discussion</td>
<td>55</td>
</tr>
<tr>
<td>5</td>
<td>ReWiRe: Middleware for Pervasive Applications</td>
<td>57</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>57</td>
</tr>
<tr>
<td>5.2</td>
<td>Open Services Gateway Initiative</td>
<td>59</td>
</tr>
<tr>
<td>5.3</td>
<td>ReWiRe framework</td>
<td>61</td>
</tr>
<tr>
<td>5.4</td>
<td>Creating pervasive applications using ReWiRe</td>
<td>66</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Creating an application ontology</td>
<td>67</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Creating services</td>
<td>69</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Creating user interfaces</td>
<td>71</td>
</tr>
</tbody>
</table>
## Contents

5.5 From task to user interface ................................. 73  
5.6 ReWiRe tools ................................................. 74  
5.7 Case studies ................................................... 75  
  5.7.1 Tux Racer ................................................. 76  
  5.7.2 SPOT applications ....................................... 80  
5.8 Discussion ..................................................... 83  

6 PerCraft: Crafting Observable Applications 85  
  6.1 Introduction ................................................... 85  
  6.2 PerCraft design methodology and tools ................. 86  
    6.2.1 Step 1: Modelling ..................................... 87  
    6.2.2 Step 2: Development .................................. 89  
    6.2.3 Step 3: Deployment .................................... 89  
    6.2.4 Step 4: Testing ........................................ 92  
  6.3 Case study: Globetrotter .................................. 93  
  6.4 Discussion ................................................... 95  

7 Semantic Web Applications 97  
  7.1 Introduction ................................................... 97  
  7.2 OWL notation in JSON .................................... 99  
  7.3 SemSon Web toolkit ....................................... 101  
    7.3.1 Importing and querying ontologies ................. 101  
    7.3.2 Creating and validating individual objects ....... 102  
  7.4 Case study: semantic user profiles ..................... 104  
  7.5 Discussion ................................................... 105  

III Interacting with Pervasive Applications 107  

8 Meta-User Interfaces 109  
  8.1 Introduction ................................................... 109  
  8.2 Universal remotes ......................................... 111  
  8.3 A view on the environment ............................... 113  
    8.3.1 End-user’s view ...................................... 114  
    8.3.2 Developer’s view ..................................... 114  
  8.4 Reference framework ..................................... 115  
    8.4.1 Environment model .................................... 115  
    8.4.2 Functional requirements .............................. 115  
  8.5 Runtime use .................................................. 117  
    8.5.1 Integrating and configuring resources ............... 117
## CONTENTS

8.5.2 Proactive behaviour ........................................ 118
8.6 Case study: ReWiRe meta-UI .................................. 119
8.7 Discussion .................................................. 121

9 Pervasive Maps ...................................................... 123
9.1 Introduction ...................................................... 123
9.2 Modelling the environment using photos .......................... 124
  9.2.1 Integrating places, sights and resources ....................... 126
  9.2.2 Integrating applications .................................. 128
  9.2.3 Tag and search ............................................. 129
9.3 Exploring the environment ...................................... 130
  9.3.1 Application-based navigation ................................. 130
  9.3.2 Resource-based navigation ................................ 131
  9.3.3 Location and orientation tracking .......................... 133
  9.3.4 Augmented reality ......................................... 134
9.4 Pervasive Maps framework ...................................... 135
  9.4.1 PM host .................................................... 136
  9.4.2 PM client .................................................. 136
  9.4.3 PM toolkit ................................................ 137
9.5 Case studies ..................................................... 137
  9.5.1 Museum visit .............................................. 137
  9.5.2 Digital scores ............................................. 139
9.6 Discussion .................................................. 141

10 Conclusions ....................................................... 143
10.1 Summary ..................................................... 143
10.2 Future research ............................................... 145
10.3 Scientific Contributions and Publications ...................... 147

A Samenvatting (Dutch summary) ................................ 153

Bibliography ....................................................... 170
List of Figures

1.1 Models and frameworks that contribute to the realization of intelligent environments ........................................... 5
2.1 Example of an OWL ontology ................................................. 13
2.2 Upper environment and domain ontologies ................................. 15
2.3 Things of interest in a pervasive environment ............................. 16
2.4 Resources are identified by a URI ........................................... 16
2.5 User and device concepts ...................................................... 17
2.6 Example of a task concept .................................................... 20
2.7 Service and user interface groundings ...................................... 22
2.8 Example of a grounded service and user interface ......................... 23
2.9 Reference architecture for the environment model ....................... 24
2.10 Example of a resource reference ........................................... 25
2.11 Aggregators and sensors ...................................................... 26
2.12 Example of a resource with an aggregator ................................. 26
2.13 Distributed query answering ................................................. 28
2.14 Example of a resource with a sensor ....................................... 29
2.15 Convenience sensors .......................................................... 29
3.1 Behaviour ontology .......................................................... 34
3.2 Example of a behaviour rule ................................................. 35
3.3 Behaviour rule groundings ..................................................... 36
3.4 Reference architecture for the behaviour model ......................... 37
3.5 Inspecting and correcting the behaviour of reactive applications .... 40
4.1 W2P architecture ............................................................ 45
4.2 W2P HTTP binding .......................................................... 48
4.3 Message exchange patterns .................................................. 50
4.4 Monitoring the message flow ................................................. 53
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5</td>
<td>Case study: media player operated through W2P.</td>
</tr>
<tr>
<td>5.1</td>
<td>OSGi architecture.</td>
</tr>
<tr>
<td>5.2</td>
<td>ReWiRe architecture.</td>
</tr>
<tr>
<td>5.3</td>
<td>ReWiRe client user interface.</td>
</tr>
<tr>
<td>5.4</td>
<td>Mapping an application ontology on an API.</td>
</tr>
<tr>
<td>5.5</td>
<td>Creating a service using ReWiRe.</td>
</tr>
<tr>
<td>5.6</td>
<td>Creating a user interface using ReWiRe.</td>
</tr>
<tr>
<td>5.7</td>
<td>Example of a user interface discovering services at runtime.</td>
</tr>
<tr>
<td>5.8</td>
<td>From task to user interface.</td>
</tr>
<tr>
<td>5.9</td>
<td>Runtime tools supported by a ReWiRe client.</td>
</tr>
<tr>
<td>5.10</td>
<td>Case study: Tux Racer.</td>
</tr>
<tr>
<td>5.11</td>
<td>Playing Tux Racer using heterogeneous devices.</td>
</tr>
<tr>
<td>5.12</td>
<td>Case study: SPOT applications.</td>
</tr>
<tr>
<td>5.13</td>
<td>SunSPOT and user interface for monitoring its sensor values.</td>
</tr>
<tr>
<td>5.14</td>
<td>Behaviour script that connects a SunSPOT and a light.</td>
</tr>
<tr>
<td>6.1</td>
<td>Globetrotter application.</td>
</tr>
<tr>
<td>6.2</td>
<td>Application model for Globetrotter.</td>
</tr>
<tr>
<td>6.3</td>
<td>Deploying peasant and wizard services.</td>
</tr>
<tr>
<td>6.4</td>
<td>Analyzing the flow of context information in an application.</td>
</tr>
<tr>
<td>7.1</td>
<td>Mapping OWL to JSON.</td>
</tr>
<tr>
<td>7.2</td>
<td>SemiSon architecture.</td>
</tr>
<tr>
<td>8.1</td>
<td>The many remotes that populate our homes.</td>
</tr>
<tr>
<td>8.2</td>
<td>Examples of meta-UIs.</td>
</tr>
<tr>
<td>8.3</td>
<td>ReWiRe meta-UI.</td>
</tr>
<tr>
<td>9.1</td>
<td>Visual markers and a device with camera give rise to u-Photos.</td>
</tr>
<tr>
<td>9.2</td>
<td>PM domain ontology.</td>
</tr>
<tr>
<td>9.3</td>
<td>PM editor and environment model.</td>
</tr>
<tr>
<td>9.4</td>
<td>Tagging resources and applications using WordNet.</td>
</tr>
<tr>
<td>9.5</td>
<td>Organizing applications by their tags.</td>
</tr>
<tr>
<td>9.6</td>
<td>Menus and photos provide access to embedded resources.</td>
</tr>
<tr>
<td>9.7</td>
<td>Pervasive Maps architecture.</td>
</tr>
<tr>
<td>9.8</td>
<td>Case studies: museum environment and digital scores.</td>
</tr>
<tr>
<td>10.1</td>
<td>Relations between developed models and frameworks.</td>
</tr>
<tr>
<td>10.2</td>
<td>Overview of selected publications.</td>
</tr>
<tr>
<td>2.1</td>
<td>Query selecting the state of a light resource.</td>
</tr>
<tr>
<td>2.2</td>
<td>Subquery selecting references to resources whose context is required to evaluate a main query.</td>
</tr>
<tr>
<td>4.1</td>
<td>Dispatching incoming messages via rules to dedicated handlers.</td>
</tr>
<tr>
<td>4.2</td>
<td>Converting a W2P message to a native object.</td>
</tr>
<tr>
<td>4.3</td>
<td>Executing a play request in a plugin for a media player.</td>
</tr>
<tr>
<td>5.1</td>
<td>OSGi manifest.</td>
</tr>
<tr>
<td>5.2</td>
<td>OSGi Activator class.</td>
</tr>
<tr>
<td>5.3</td>
<td>Sharing and accessing distributed resources on a ReWiRe client.</td>
</tr>
<tr>
<td>5.4</td>
<td>Fetching the properties of a resource.</td>
</tr>
<tr>
<td>5.5</td>
<td>Importing an application ontology and querying resources.</td>
</tr>
<tr>
<td>5.6</td>
<td>Interacting with a remote service using a proxy component.</td>
</tr>
<tr>
<td>5.7</td>
<td>Query asking for devices that can serve a Tux Racer game.</td>
</tr>
<tr>
<td>6.1</td>
<td>Query asking for the current location of the user.</td>
</tr>
<tr>
<td>7.1</td>
<td>Creating a FOAF user profile using SemSon.</td>
</tr>
<tr>
<td>9.1</td>
<td>Initiating DigiScores.</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Contents

1.1 Background .......................................................... 1
1.2 Problem statement .................................................... 2
   1.2.1 Merging real and digital worlds .......................... 2
   1.2.2 Merging user and developer roles ....................... 3
1.3 Towards do-it-yourself pervasive applications ................. 4

1.1 Background

Pervasive computing, also referred to as ubiquitous computing, is an effort towards achieving the vision of Weiser where devices seamlessly integrate into people’s everyday life [Weiser 91]. Appliances should vanish into the background to make users and their tasks the central focus rather than computing devices and technical issues. A place that is augmented with networked devices able to monitor users and environmental resources, is called a pervasive (computing) environment. The applications deployed in such an environment are denoted as pervasive applications and are typically distributed amongst a set of heterogeneous devices. For example, a domestic pervasive environment might interconnect lighting and environmental controls with sensors woven into clothing so that illumination and heating conditions in a room can be adjusted automatically. Another common scenario posits refrigerators ‘aware’ of their suitably-tagged contents so that they are able to propose a variety of
menus from the food actually on hand and warn users of stale or spoiled food.

1.2 Problem statement

In this work, we explore two trends that are caused by the paradigm of pervasive computing. First, real and digital worlds become mixed up when daily environments are augmented with embedded computing devices. Not only the environment should become aware of the user, but the user should also become aware of the environment’s digital dimension. Second, with an increasing amount of technology surrounding us, user and developer roles get intertwined when it comes to the configuration of applications. Developers can not deliver pre-configured applications because it is always runtime in a pervasive environment. Mobile devices enter and leave an environment at runtime and can be used by end-users to interact with their surroundings.

1.2.1 Merging real and digital worlds

Real and digital worlds readily meet each other in everyday consumer products: a television is a physical device, but it is operated and configured using digital user interfaces. Embedded software is no longer a means to make a device’s buttons magically react when they are pressed. Instead, it presents all of a product’s features in a digital way by making use of on screen displays that correspond to a device’s user interface. This allows products to act smarter and better integrate with our daily life. A notable example of this tendency is to provide access to personal and interactive content such as photos, movies, games and e-mail. An environment (e.g. a house or office place) can also be augmented with digital services that sense the user’s activities and configure the environment to better support their tasks. Typical scenarios arise from the health sector where elderly people are assisted by technology in their daily activities. For example, applications remind them of taking their pills, prevent fire when the stove is left on after cooking and turn down the lights when their favourite television soap starts.

When more and more products become smart and services get invisibly embedded in homes and public spaces, mobile devices such as phones and ultra-mobile PCs become increasingly important to get in touch with our surroundings, from anywhere, anytime. Besides, handheld devices can provide the environment with information about a user’s availability (e.g. in a meeting) or current location which that is derived from a GPS chip. This leads to a flow of data sensed by devices in the physical world that is synthesized and
processed by software components in the digital world, also known as context information. Dey et al. refer to context as follows [Dey 01]:

any information that can be used to characterize the situation of entities (i.e., whether a person, place or object) that are considered relevant to the interaction between a user and an application, including the user and the application themselves.

One of the major challenges in the design of pervasive applications is to manage, collect and share context information that is produced at runtime. Context-aware applications react to context updates such as a change in the user’s location or room temperature by automatically adapting to the new situation. The slightest change in the environment configuration can trigger an application to adapt and might result in new context updates. The first objective we address in this work is to

cope with dynamic pervasive environments by connecting context information and physical objects so that changes in the digital world are reflected in the real world and vice versa.

This brings up several questions for which we try to find answers, assisted by previous research efforts. For example, how can we build applications that can operate in an environment of which the configuration is not know beforehand? And, how can we help users to predict and understand the consequences of performing an action in either the digital or real dimension of a pervasive environment?

1.2.2 Merging user and developer roles

When our surroundings become integrated computing systems, this surfaces both opportunities and pitfalls. On the one hand, technology can improve the way we interact with the environment, but on the other hand privacy issues and failures caused by a misunderstanding between user and machine can make end-users reluctant to new technology. For example, a computing system that refuses to open the garage port because it does not recognize the user’s new mobile phone as trusted device will result in frustration. Besides, an application that is updated to a newer version (e.g. remotely by a product’s manufacturer) might become incompatible with other appliances in the environment. Such situations can arise because computing systems are not all knowing and a software developer can not envision all possible circumstances in which an application will be used. Manual configuration is often indispensable to avoid mismatches between user preferences and the actual system
behaviour. To create self-explanatory interactive environments, both developers and users will need to abandon their traditional roles of developing core software components and interacting with individual applications. Developers should also pay attention to tools that assist end-users in the configuration and assembly of smart products and to help them observe an application’s current state. By adopting general tools that integrate with any type of pervasive environment, users can step beyond their role of interactor and evolve to programmers of their own environments. As a second objective in this work we explore paths to

make users aware of the technology that is surrounding them and provide tools to interact with the digital dimension of a pervasive environment and configure its behaviour.

1.3 Towards do-it-yourself pervasive applications

The challenges outlined in the previous sections give rise to do-it-yourself pervasive applications, allowing users to actively participate in their deployment and configuration.

Figure [1.1] depicts the structure of this dissertation: three parts elaborate on models, development and deployment frameworks and interaction tools. We contribute to the creation of do-it-yourself applications through a combination of research efforts in different related domains. For each of these domains, we propose building blocks for developers and users that incrementally give rise to ambient intelligent environments.

The first challenge, coping with dynamic pervasive environments by connecting context information and smart products so that changes in the digital world are reflected in the real world and vice versa, is addressed via models and development frameworks. Models are described in a first part of this dissertation and correspond to a description of one or more entities represented in a computing system. For example, a user model could describe a person’s identity and friends. We propose two generic models, namely an environment model in chapter 2 and a behaviour model in chapter 3 which respectively capture the context of use of a pervasive environment (who is online, which devices are available?) and the behaviour of running applications (what should happen when a user walks through the door?). We call these models semantic because they are built using ontologies which create a shared understanding between user and system. In addition to models, we contribute to the first challenge with a number of frameworks in a second part of this
Figure 1.1: Models and frameworks for creating do-it-yourself pervasive applications contribute to the realization of intelligent environments where users and system can seamlessly interact with each other.

dissertation. These frameworks provide a reusable software base that assists developers in building pervasive applications. In chapter 4 we present W2P, a light-weight framework allowing resources that inhabit a pervasive environment to exchange messages. Since physical and computational resources are typically distributed in such environment, a flexible communication middleware is key to make them talk over a network. Furthermore, the ReWiRe framework, discussed in chapter 5, provides a middleware platform on which components can be deployed that make up a pervasive application such as services or user interfaces. ReWiRe links the deployment of these components with the context of use and thus interconnects real and digital worlds. The PerCraft design strategy outlined in chapter 6 further exploits the capabilities of ReWiRe to create pervasive applications that can be inspected at runtime using integrated tools. Chapter 7 ends the second part with an overview of the SemSon toolkit which brings context to the Web browser. The Web and its applications play an increasingly important role with the advent of powerful mobile and Web-enabled devices. When semantics reach the Web browser, client-side Web applications can become aware of their context and seamlessly integrate in a pervasive setting.
The second challenge, making users aware of the technology that is surrounding them and providing tools to interact with the digital dimension of a pervasive environment and configure its behaviour, aims to integrate better users and their computer augmented environment. The third part in this dissertation is devoted to this objective. We present meta-user interfaces in chapter 8 as a pervasive variant of a ‘start menu’ on desktop computers. A meta-user interface allows users to explore their surroundings and find out about the tasks that are supported. When combined with middleware such as ReWiRe, we show that the meta-user interface can be used to request migratable user interfaces and runtime tools to configure and interact with resources and applications embedded in the environment. Chapter 9 further develops the concept of meta-user interfaces with Pervasive Maps. In this framework, photos are used to represent a pervasive environment and interact with it.

We conclude this dissertation in chapter 10 with closing remarks, possible directions for future research and an overview of scientific contributions and publications as a result of this work.
Part I

Modelling Pervasive Applications
Chapter 2

Semantic Environment Model

Contents

\textbf{2.1} Introduction .............................................. 9
\textbf{2.2} Ontologies ................................................. 11
\hspace{1em} 2.2.1 Web Ontology Language ......................... 12
\hspace{1em} 2.2.2 Upper and domain ontologies .................... 14
\textbf{2.3} Environment ontology .................................... 14
\hspace{1em} 2.3.1 Resources ............................................ 14
\hspace{1em} 2.3.2 Users and devices .................................... 16
\hspace{1em} 2.3.3 Services .............................................. 17
\hspace{1em} 2.3.4 Tasks and user interfaces ............................ 19
\hspace{1em} 2.3.5 Groundings ........................................... 21
\hspace{1em} 2.3.6 Domain concepts .................................... 23
\textbf{2.4} Distributed context management ....................... 24
\hspace{1em} 2.4.1 Context on demand: aggregators .................... 25
\hspace{1em} 2.4.2 Context on change: sensors .......................... 28
\textbf{2.5} Discussion .................................................. 29

2.1 Introduction

In this chapter we describe a model to capture the context of a pervasive environment. The main purpose of this model is to ease the creation of context-aware applications by allowing software components to query the current state of the environment. One of the requirements identified by Grimm et al. to provide system support for pervasive applications is to embrace
contextual change and not hide it from applications [Grimm 04]. This demands for a model that can deal with a changing context of use and inform applications of these changes. Gu et al. [Gu 04b] classify existing context models for intelligent environments in three categories: application-oriented (context is only represented for specific applications), conceptual model-oriented (conceptual representation of context using ER and UML diagrams) and ontology-oriented approaches (strong focus on knowledge sharing and context reasoning). Due to a lack of a formal basis in application-oriented approaches and a lack of support for knowledge sharing and reasoning in model-oriented approaches, the authors conclude that an ontology-oriented approach is most suitable for creating context-aware pervasive applications. Several ontology-based models have been proposed to represent context [Ye 07, Chen 04b, Gu 04b, Preuveneers 04]. Chen et al. [Chen 04b] defined an ontology for pervasive applications, called SOUPA. It is expressed in OWL and combines existing vocabularies in a shared ontology that covers the main concepts in pervasive computing. Gu et al. [Gu 04b] stress the importance of an extensible context model and define quality parameters they use to characterize data. Their SOCAM framework contains a single ontology that defines concepts of activity, location, person, time, computational entity, etc. Preuveneers et al. [Preuveneers 04] also proposed an extensible ontology for ambient intelligent environments, defining several upper level concepts. These ontologies are mainly conceptual models geared towards the creation of context-aware systems, while our model focuses on runtime support and practical use.

Inspired by previous research efforts, we define a set of design criteria that help to understand the architecture of our semantic environment model:

- Open-ended: Since a pervasive environment is not limited to a specific usage scenario, a model representing its context must be generic enough to deal with different types of pervasive environments on the one hand, but also support a sufficient level of detail on the other hand.

- Dynamic: The model must be able to cope with multi-purpose environments that evolve while the system is being used. Apart from hard- and software entities that might join or leave an environment at runtime, the entire context of use of an environment can change. For example, the role of a room could evolve from meeting room to cinema room, supporting different activities and hence requiring a different configuration.

- Queryable: The ability to efficiently acquire specific context data from
anywhere in the environment’s network is a key feature of the environment model. Pervasive applications depend on fast and reliable context retrieval to make decisions based on the current state of the environment.

- **Distributed**: Since a pervasive environment can grow large – consider for example an airport equipped with pervasive computing systems – storing all context information in a central place would be highly inefficient. Centralizing context would generate a huge amount of network traffic in order to replicate data produced by different connected computing devices. Instead, we propose a distributed model which encompasses all context-generating computing devices.

We address these criteria with a combination of semantic Web technologies (i.e. ontologies) and a distributed architecture that simplifies the exchange of context information at runtime. Our context model differs from existing approaches by directly linking concepts defined in the model with software components. This is achieved via *groundings* that link a service concept with its implementation (see section 2.3.5) and reference concepts pointing to data objects (e.g. Java objects) that hold the actual context information (see section 2.4).

### 2.2 Ontologies

Ontologies are used to represent knowledge (structured information) that can be understood both by humans and machines. An often cited definition of an ontology is the one of Gruber [Gruber 93]:

> An ontology is a formal, explicit specification of a shared conceptualization.

The word ‘formal’ hints a precise mathematical description, the term ‘explicit’ denotes that concepts and their relationships are clearly defined and ‘shared’ refers to the existence of an agreement between ontology users and software agents [Fensel 01]. The use of ontologies has become common on the Web, for example to (automatically) categorize a website such as Yahoo! [Labrou 99] or to create taxonomies that categorize products and their properties on a site like Amazon which can then be processed by software agents. Similar, we use ontologies to describe context information in a pervasive environment which can then be processed by pervasive applications.
We opted for ontologies and in particular the Web Ontology Language (OWL) \cite{OWL04} because of different reasons. First, ontologies are a step forward to the semantic Web. A clear evolution towards networked environments in which the Web is entangled can be noticed \cite{Kranz10} and leveraging the technologies used to develop (semantic) Web applications improves compatibility with this type of applications. Second, ontologies are subject of research for several years and various conferences in the domain of knowledge engineering and other branches of computer science (ubiquitous computing amongst others) are even specifically devoted to ontologies. This not only resulted in tool support (e.g. Protégé\cite{http://protege.stanford.edu/}, Jena\cite{http://jena.sourceforge.net/}, Pellet\cite{http://pellet.owldl.com/}), but also in the use of ontologies in applied domains such as bioinformatics (e.g. the Gene Ontology\cite{http://www.geneontology.org/}). Third, OWL is a standardized specification to express ontologies and is adopted by many tools.

### 2.2.1 Web Ontology Language

The Web Ontology Language (OWL) \cite{OWL04} is a semantic markup language for publishing and sharing ontologies and is developed as a vocabulary extension of the Resource Description Framework (RDF) \cite{RDF}. An OWL ontology consists of classes, individuals and properties. Classes place constraints on sets of individuals in a universe of things (e.g. things inherent to a Web document, application or environment). Instances are individuals that belong to one or more classes and maintain so called ‘is-a’ relationships with these classes. A property is a binary relation of which two types are distinguished: datatype properties define relations between instances of classes and RDF literals and XML Schema datatypes and object properties define relations between instances of two classes. In summary, OWL can be used to:

1. formalize a domain by defining classes and properties of those classes,
2. define individuals and assert properties about them, and
3. reason about these classes and individuals to the degree permitted by the formal semantics of the OWL language.

OWL provides three increasingly expressive sublanguages: OWL Lite, OWL DL and OWL Full. We use the OWL DL sublanguage, as it contains
the necessary expression power for the applications discussed throughout this text. Unlike OWL Lite, OWL DL allows to express class equivalence which is a useful feature to relate similar independent concepts when dealing with multiple ontologies. OWL DL is named so due to its correspondence to description logics [Baader 03]. It supports maximum expressiveness without losing computational completeness and decidability of reasoning systems, which can infer additional information based on data that is explicitly provided. When applications need even more expressiveness, OWL Full semantics can be incorporated in the model without limitations.

Figure 2.1: Fragment of an OWL ontology which can be used to describe a car park (a) and an instance of a specific car (b).

Figure 2.1 shows an excerpt of an OWL ontology that can be used to categorize the car park of a taxi company. The filled rounded rectangles correspond to OWL classes and transparent rectangles to instances of classes (individuals). In our notation, isa relationships denote either an ‘instance of’ relation between an instance and a class (rdf:type) or a ‘subclass of’ relation between two classes (rdfs:subClassOf). In the example, the mileage and plate relations indicate that each Car has a mileage and (license) plate datatype property. Note that the BMW_1 instance is also an instance of the Vehicle class due to its class hierarchy. If drivers of the taxi company have their own taxi, we can introduce a Driver class and drives object property that interconnects a driver and a car. Although the OWL language is much more expressive than depicted in this schema (e.g. cardinality constraints on properties are omitted), we consider the granularity of detail provided in this schema and similar schemes throughout this text in balance to understand these schemes whilst not overloading them with unnecessary details.
2.2.2 Upper and domain ontologies

Ontologies are useful to create an open-ended environment model since different ontologies can be merged. We can combine generic concepts which are typically defined in an upper ontology with more specific concepts which are part of a domain ontology. Consider for example the Suggested Upper Merged Ontology (SUMO) [Niles 01]: it consists of a comprehensive list of generic concepts such as Animal and Vehicle which are described in more detail in domain ontologies. These domain ontologies describe specific domains such as finance, people or transportation and extend the knowledge base of the upper ontology. Likewise, we can identify generic concepts that are applicable to all kind of pervasive environments, unite them in an upper ontology and merge these with domain ontologies that provide more detail about a specific environment or application. In our environment model, upper and domain ontologies are dynamically aggregated and hence make up a scalable knowledge base that can be adapted to any target environment, even at runtime when the context of use changes. Figure 2.2 depicts an example of our aggregated environment ontology. The upper ontology with namespace http://edm.org/environment# is a default part of our environment model, while domain ontologies that extend the upper ontology such as http://edm.org/environment/light# are only plugged in when needed. The next section elaborates on this environment ontology and in particular on its upper ontology.

2.3 Environment ontology

The heterogeneous nature of a pervasive environment complicates the description of things of interest. Many terms used to describe such an environment are actually used in different contexts. For example, the term service might refer to application logic, but often it is also used to depict a user interface for interacting with application logic. To avoid such ambiguity, we provide a detailed description of the concepts that populate our environment model and their intended use in the remainder of this section.

2.3.1 Resources

We use the term resource to denote anything of interest in the environment, ranging from software resources to hardware resources and end-users. Some typical resources that inhabit a living room are shown in figure 2.3: a television panel, a media service, etc. The properties of the resources indicated on the
Figure 2.2: Aggregating an upper ontology with domain ontologies dynamically integrates the required semantics in our model so that any target environment can be described.

The Resource concept shown in figure 2.4 acts as a base class to describe resources in our environment ontology. Since OWL is used to express this ontology, a resource is defined as an OWL Class which on its turn is an OWL Thing as defined in the OWL specification [OWL 04]. Instances of the Resource class, denoted as individuals in the OWL specification, correspond to the current set of resources that make up the pervasive environment. Each class and individual is identified by a unique URI, dictated by the RDF framework [RDF] on which OWL is built. For example, the URI [http://edm.org/environment#Resource] identifies a concept with RDF namespace [http://edm.org/environment#] and RDF ID Resource, alias our Resource class. To make sure that a resource can be presented in a human
Figure 2.3: Things of interest in an environment are modelled as resources in a digital representation of the environment.

readable way, we attached name and icon properties.

Figure 2.4: Resources are identified by a URI and have a name and icon.

2.3.2 Users and devices

Many resources play an important role in an interactive pervasive environment. Amongst these, we consider users and devices the most generic ones and hence included them by default in our environment model. With the term user we refer to a human person (i.e. not a user agent such as a Web browser) who can interact in a pervasive environment. The ability to identify users in a pervasive environment is indispensable for several reasons. Applications that rely on security and access control mechanisms (e.g. parental control) or applications that need to distinguish between users’ interaction in a multi-user environment depend on user awareness. Recognizing users is also a prerequisite for a pervasive computing system aiming to deliver a personalized interaction experience (i.e. by adapting its behaviour to user preferences). Furthermore, devices are major resources because without them, there is simply no pervasive computing environment. Computing devices (or devices in short) can
2.3 Environment ontology

fulfill a double role: they provide hardware and an operating system to run pervasive applications and/or are used as a medium to interact with pervasive applications. When used as an interaction device, a device is likely to be accessible in the environment (e.g. a cell phone or embedded touch-panel), but when embedded in e.g. a kitchen appliance, the computing device disappears in the surroundings. The specific roles of a device are typically determined by a pervasive application based on a description of the capabilities of the device, including its input and output mechanisms and its ability to run services? We consider similar device and also user properties to be stored in a device profile such as CC/PP [CC/PP 04] (e.g. properties of a device’s hardware and software platform) or a user profile such as FOAF [FOAF 10] (e.g. friend relations).

Figure 2.5: User and device properties are not predefined, but can be integrated in the model using dedicated profiles.

In the environment ontology, users and devices are represented by the User and Device concepts as depicted in figure 2.5. These concepts mainly serve as an umbrella to which additional information can be attached, for instance properties that are stored in a user or device profile. Note that such a profile can be aggregated with the environment ontology or referred to via a property. The former assumes that the profile itself is described by an ontology, while the latter expects a profile to be defined in an external file with dedicated format. The usedBy relation (and its inverse uses relation) connects a user and an (interaction) device. This allows a pervasive application to address a user instead of a device, and to derive the current interaction device(s) of a user from the environment model.

2.3.3 Services

We use the term service to refer to the functional building blocks of a pervasive application. Services are functional components created by developers and run on the available devices. They expose a software interface so that other
software components (e.g. another service or a user interface) can interact with them. Different service types can be recognized:

- **Web services** are primarily used as a means to interconnect their information systems over the Internet and to make business available to clients. These services are hosted on a Web server and remotely executed by a client using a SOAP [SOAP 00] or REST [Richardson 07] API. Open standards such as WSDL [WSDL 01] and OWL-S [OWL-S 04] feature the description of Web services to simplify interaction with a service, for example using a proxy component that is automatically generated from a WSDL file. RESTful APIs such as the Google Search API[^5] gain in popularity because they allow light-weight interaction with Web services from within a Web browser using AJAX technologies.

- **UPnP [UPnP] and Jini [Jini]** services are typically published on a local network (e.g. home network) and make use of discovery protocols based on multicast. Unlike Web services, they often embed a user interface so that end-users can directly interact with them once they are discovered. UPnP services are for instance embedded in routers and multimedia appliances, which can then be operated and configured from UPnP-compatible devices such as smartphones.

- **OSGi [Alliance 09]** services are Java object instances, registered into an OSGi framework such as Apache Felix[^6] or Knopflerfish[^7]. These services are encapsulated in OSGi bundles which can be remotely installed, started, stopped, updated and uninstalled at runtime (life cycle management). They are typically used when dynamic deployment of software components on devices is envisioned without requiring restarts.

- **Many other types of services** have been proposed for use on embedded, special-purpose devices.

We make no differentiation between types of services in the environment ontology: instances of a `Service` class can be any type of service, independent of the technology that was used to develop the service. In section 5.4.2 we illustrate how an arbitrary service can be created, packaged and deployed. A grounding (see section 2.3.5) relates a `Service` concept with programming code by describing the technology that was used to build the service (Web services, UPnP, OSGi, ...).

2.3 Environment ontology

2.3.4 Tasks and user interfaces

With the term user interface – and the corresponding UI concept in the ontology – we denote a component for end-users to interact with a service and to observe its state. Various approaches have been suggested to create user interfaces for pervasive applications:

- Manually designed user interfaces provide the best aesthetic quality and can make effective use of the input capabilities of a target device. Even though designers can make use of layout managers that adapt the user interface to the screen size at hand, a user interface leverages toolkits which may not be available on each target platform. Hence the cost of designing and maintaining manually designed user interfaces for different platforms is high. Tools such as Gummy [Meskens 08] and Jelly [Meskens 10] try to lower the burden of designing multi-device user interfaces by eliminating the need to switch between different design tools and by providing tool support for keeping the user interfaces consistent across different platforms and toolkits.

- Generated user interfaces on the other hand, can dynamically adapt to the target device, user preferences and other environmental context properties. A common idea in frameworks that (semi-)automatically generate user interfaces such as SUPPLE [Gajos 04] and the Personal Universal Controller (PUC) [Nichols 06] is the use of an abstract description that defines the functionality the interface should expose to the user. The presentation of those features is left to a renderer which typically uses a constraint-based layout algorithm to present the user interface on a certain display.

A stringent difference between user interfaces for desktop application and user interfaces for pervasive applications is the heterogeneity of (mobile) devices that can be used to interact with the latter type of application. This variety of interaction devices demands for appropriate user interfaces that match the modalities (e.g. graphics or speech) and constraints (e.g. tiny or large display) of the target interaction device. Depending on the device at hand and goal to accomplish, the best option could be to automatically generate a speech user interface or to render a high-quality manually designed graphical user interface. While it is likely that the quality of generated user interfaces will improve in the future, we witness that companies like Google provide native applications for their services, next to a Web interface which is
readily optimized for mobile devices. This indicates that there is still a need for user interfaces tailored to specific devices to improve the user experience.

To support different types of user interfaces we introduce the term *task* which denotes a conceptual entity that describes an activity the user can undertake in the environment. In our definition, a task is merely a description such as “play music” or “operate lights” that provides insight in the features of a pervasive environment for the user. Unlike task models such as discussed in [Paterno 02], the **Task** concept in our environment ontology does not describe transitions in an application’s user interface, but rather represents a goal that can be accomplished through interaction with a user interface component. A task maintains a relationship with user interfaces representing the task and services it depends on as shown in figure 2.6. Note that the **Light**, **LightService**, **AnyLightService**, **SwingUI**, **LightSwingUI** and **ToggleLightTask** concepts in this example are not part of the upper environment ontology, but are provided by domain ontologies (see also figure 2.2). The **ToggleLightTask** in figure 2.6 depends on an instance of a **LightService**.

![Figure 2.6: Example of a task, supported by a resource, that depends on a service and that can be presented by a Swing user interface.](image)

The **AnyLightService** instance indicates that any available **LightService** running in the environment can be used to execute this task. The **LightSwingUI** is a candidate user interface to present the **ToggleLightTask** on an end-user device. Other user interfaces such as a **LightSpeechUI** can be attached to the task as well to support different modalities and platforms. Since tasks often arise from specific resources, we support the option to connect resources and the tasks they support in our environment model. For instance, by linking the **Light** and **ToggleLightTask** concepts via a **supportsTask** relationship we can specify that each instance of a light resource supports the task of turning it on or off.
2.3 Environment ontology

2.3.5 Groundings

The Service and UI concepts in the environment ontology represent software components. In order to interact with a service component or to render a user interface component, a device must become aware of the component’s technology (e.g. its supported toolkits and communication protocols). The binding between model and component is realized through groundings: a grounding defines a contract for using a software component. This contract is interpreted by a device and if understood and considered compatible, the device can make use of the component.

Groundings are defined as concepts in a domain ontology: services and user interfaces are grounded by a ServiceGrounding and UIGrounding respectively as depicted in figure 2.2. For each type of service we summarized in section 2.3.3 (Web services, UPnP services, OSGi services), we define a separate grounding:

- A grounding for Web services links a Service with a WSDL document and optionally uses OWL-S concepts to describe its methods in more detail.

- A grounding for UPnP services specifies properties such as descriptionURL and controlURL to interact with a UPnP service.

- A grounding for OSGi services relates a Service with an OSGi bundle (or OSGi service inside this bundle). It defines properties such as bundleLocation and bundleVersion to describe an OSGi component and hence enable remote deployment of the bundle.

- Other groundings such as a link to a dynamic library that embeds application logic can be created as well.

In fact, a service can be grounded in different ways at the same time so that it can be either deployed as a Web service or UPnP service, depending on a device’s operating system and capabilities. Likewise, user interfaces can be grounded using different technologies and modalities. A grounding for a Web interface simply points to a website and a grounding for a speech interface can link to a VoiceXML document. For each markup language that is used to describe a user interface, we can create a separate grounding. In addition, a coded user interface component such as a Java Swing interface can be encapsulated in an OSGi bundle and grounded similar to an OSGi service.
The ServiceGrounding and UIGrounding concepts depicted in figure 2.7 specify how to interact with a service and render a user interface respectively. For services, we distinguish between an implementation and proxy grounding:

- An implementation grounding defines a component that contains the implementation of a service. It can be used to dynamically deploy a service on a device.

- A proxy grounding refers to a component that provides an API for remotely interacting with a service. Proxy groundings are particularly useful to connect user interface components with application logic.

Figure 2.7: Groundings connect services and user interfaces with software components that contain their implementation and are interpreted to interact with a service or render a user interface.

Using these groundings we can support a wide variety of technologies to interface with a software component and hence guarantee a high scalability of our environment model and the computing environment it is deployed in. For example, a UPnP grounding can link a service with its UPnP service URL, so that a UPnP capable device can directly interact with this service by analyzing its grounding. Another example is an OSGi grounding that links a service or user interface with an OSGi bundle containing its implementation, illustrated in figure 2.8.

Executing a task then corresponds to selecting a suitable user interface for the task. This is achieved by comparing the capabilities of the device at hand with the groundings of available UI instances for the task. Service dependencies indicate whether a task can be executed in the first place: if no instance of a required service is available in the environment, the task can not be executed. The process of rendering a user interface for a task and allocating resources for it (e.g. connecting it with remote services) is further elaborated on in section 5.5.
2.3 Environment ontology

Figure 2.8: Example of a service and user interface grounded by means of an OSGi bundle.

2.3.6 Domain concepts

Most concepts discussed so far (resources, users, devices, services, ...) are generic concepts which play a role in almost any pervasive environment and therefore are part of an upper ontology in our environment model. Since more detail is often needed to describe the context of specific applications and resources, additional concepts defined in a domain ontology can be merged with the concepts that already exist in the environment model. In fact, generic concepts such as the Resource and Service concepts serve as base classes upon which domain-specific concepts can be constructed, as illustrated in figure 2.2.

Domain ontologies can serve different purposes:

- **Application ontologies** describe concepts related to an application domain such as MediaPlayer, and Playlist.

- **Grounding ontologies** introduce technology bindings which allow to interface with software components, e.g. UPnPGrounding, OSGiGrounding, SwingUI, etc.

- **Bridging ontologies** provide concepts and relations to connect our model with external ontologies such as the WordNet lexicon [van Assem 06], i.e. to map concepts on natural language and exploit linguistic relations.

Grounding and bridging ontologies can be considered as generic extensions to the upper ontology. The upper ontology does not define other generic concepts, e.g. related to location or time, which are prevalent in various ontology-based models [Ye 07]. Therefore we also rely on generic domain ontologies to introduce such concepts and accompanying services to manage their instances and relations with resources. For example, we created a spatial ontology and
a dedicated service that is responsible for updating spatial relations between resources such as leftOf, belowOf when the position of a resource changes [Aksenov 09]. A similar approach can be taken for time: when a service keeps track of context evolution over time, applications can query the history of the environment configuration.

2.4 Distributed context management

Pervasive applications rely on the ability to collect information about resources at runtime. To accommodate this requirement, a pervasive architecture must find a good compromise between the efficiency of storing and retrieving context information. We suggest a reference architecture that spans our environment model over a set of heterogeneous computing nodes and a central repository, the context store, as shown in figure 2.9. The context store stores information about the semantics of resources such as properties of resources and relations between resources as defined in the upper and domain ontologies. Domain ontologies are imported as static documents in the context store and make up a knowledge base that is shared amongst pervasive applications.

Figure 2.9: Ontologies describing the environment topology are imported in the context store which runs on a dedicated server device (D5). Resources are published on client devices (D1, D2, D3) and advertised with a reference in the context store. Queries are directed to a query engine at the server device which collects resource-specific context data from client devices when needed in order to evaluate a query.

Besides the semantics of the environment and its applications, the context store also includes a registry of references to instances of resources whose execution context resides on distributed computing nodes. Each networked device can fulfill the role of computing node, provided that it runs an operating system that can be leveraged by pervasive applications to deploy services on. In this case, resources such as the device itself, services running on the device or a resource attached to the device are dynamically advertised with
2.4 Distributed context management

a reference in the registry that points to the resource instance (e.g. a data object) on the device, illustrated in figure 2.10. Note that a resource’s context is not duplicated in the context store, but returned in a semantic data format upon request. The context retrieval is based on the content negotiation mechanism defined in the HTTP specification [HTTP 99]: a context requester (a service) asks for a resource’s context represented as e.g. RDF triples and this format is then supplied by the server (the device that holds the resource). As such the state of a resource can be queried regardless of where and how it is stored internally, e.g. on a chip or in a database on the Web. When a service fails or a device and its local resources leave the environment without proper announcement, the context store can become polluted with deprecated references. To avoid this, we include a simple garbage collection mechanism: resources are regularly probed to verify their existence and if a resource remains unresponsive for a certain period of time, its reference is removed from the context store.

![Figure 2.10: Example of a reference to a Light resource.](image)

To use context information in a meaningful way, applications must be able to acquire context and get notified of context changes. For this purpose, we introduce aggregators and sensors that can be attached to resources as shown in figure 2.11. Inspired by the OUTPUT class defined in the OWL-S ontology [OWL-S 04], the PARAMETER TYPE of an output can be an XML schema type (e.g. string, integer, float, …), but also an OWL class (e.g. a RESOURCE) in which case a URI pointing to an instance of this class can be provided as output value. In the next sections, we further elaborate on context processing by means of aggregators and sensors.

2.4.1 Context on demand: aggregators

An aggregator collects information about a resource and delivers this data in a set of output parameters. Figure 2.12 shows an example of an aggregator: the
Resources can request context on demand through aggregators and publish context changes to other resources through sensors.

**ONOffAggregator** returns the state of a light as an integer value (e.g. 1 = On, 0 = Off). A more complex aggregator could return all Light resources with On state along with their current intensity. To acquire this information, we envision the use of queries such as the one in listing 2.1. Queries are a very important instrument to realize aggregators because they can be executed from any device in the network that has access to the context store and its query engine. Therefore we will first explain how queries can be answered over a distributed context model.

Since the entire environment context is described by means of ontologies and instances of ontologies, semantic query languages such as SPARQL and OWL-QL are suitable candidates to query the model. However, these query languages assume that all context data is centralized in a semantic database while we envision distributed storage as illustrated in figure 2.9. Federated query solutions such as DARQ [Quilitz 08] and KAONp2p [Haase 07] do exist, but rely on distributed computing nodes capable of query processing and
2.4 Distributed context management

aggregating query results. These solutions scale well in environments where
large server-side datastores are addressed, but they are costly and complex
to integrate in a dynamic heterogeneous environment. Embedded computing
devices are constrained in processing power and hence are less suited for query
processing. In order to query a distributed context of use using conventional
query languages, we propose a query engine layered on top of the context store
that processes a query $Q$ in three steps:

1. **Resource selection**: a subquery $Q_r$, derived from the final query $Q$ or
   specified manually, selects references to resources whose context should
   be resolved in order to answer $Q$.

2. **Context aggregation**: a temporary model is prepared that shares domain
   knowledge with the context store and which includes the current state
   of selected resources, fetched from distributed nodes.

3. **Query evaluation**: with all relevant context data aggregated in a tem-
   porary model the query $Q$ can be evaluated against this model.

Listing 2.2 shows a SPARQL query $Q$ that asks for the devices that are used by
User1. Figure 2.13 illustrates how this query is evaluated. Since the context
store only contains references to resources, it does not include the required
resource contexts (i.e. instance data) to evaluate the query. By analyzing
the domain of the `usedBy` predicate in the ontology, we can determine which
resource contexts need to be included in the temporary model in order to
guarantee that query results are complete. As the domain of the `usedBy`
relation defines DEVICE resources, references to devices are selected using a
query $Q_r$ and for each reference, the context of the resource it refers to is
fetched and aggregated in the temporary model. We also foresee the option
to manually specify the query $Q_r$ using the `RESOLVE` keyword. This keyword
extends the syntax of the query language that is used (e.g. SPARQL) and is
interpreted by the query engine in a preprocessing step.

$Q$ : SELECT ?d WHERE {?d :usedBy <:User1>}

$Q_r$ : SELECT ?r WHERE {?r a :ResourceRef ; :type ?t . ?t rdfs:subClassOf :Device}

Listing 2.2: The subquery $Q_r$ selects references to resources whose context is
required to evaluate query $Q$.

Aggregators hide underlying queries and can be compared with ‘getters’ in
an object-oriented programming language, returning multiple output values.
Figure 2.13: To answer a query $Q$ similar to the one in listing 2.2, the query engine first generates a subquery $Q_r$ that selects references to resources whose context is required to evaluate $Q$. This context is fetched from distributed computing nodes and aggregated in a temporary model. Finally, the query $Q$ is evaluated against this model and results are returned to the invoker ($D_1$ in the figure).

As such, an aggregator corresponds to an executable component that delivers context on demand.

2.4.2 Context on change: sensors

Resources spontaneously publish context information through sensors to which other resources can subscribe. When a context change occurs in a resource, a corresponding sensor related to the resource is triggered and resources subscribed to the sensor will be notified of the change. Similar to an aggregator, the updated context information is (optionally) exchanged in output parameters. Figure 2.14 shows an example of a sensor attached to a LIGHT resource. Note that the model excerpt in this figure is almost identical to the one in figure 2.12. Sensors are modelled likewise to aggregators, but instead of being invoked on demand, sensors are triggered upon change.

To get notified of sensor updates, software components are expected to subscribe to \{Resource, Sensor\} pairs such as \{MyLight, OnOffSensor\}. The AnySensor and AnyResource concepts ease the subscription to multiple sensors and resources at once. To filter the amount of context
Figure 2.14: Example of a Light resource with a sensor.

updates that are published to resources, we suggest additional convenience sensors. Convenience sensors extend existing sensors to support a more detailed selection of context changes at the subscription stage. For instance, the ONOFFSENSOR in figure 2.14 can be extended with an ONSENSOR and an OFFSENSOR (see figure 2.15). In this case, the ONSENSOR will be triggered as a special variant of an ONOFFSENSOR with output value 1 (corresponding to an On state) and all resources that are subscribed to either the ONSENSOR or the ONOFFSENSOR will get notified of the context update.

Figure 2.15: Convenience sensors enable fine-grained subscriptions to relevant context changes.

2.5 Discussion

In this chapter we have introduced an ontology-based model for describing the context of a pervasive environment. The model contributes to the creation of context-aware applications by allowing software components to query the current state of the environment. An upper ontology describes generic resources that inhabit a pervasive environment such as users, devices, services and tasks, while domain-specific ontologies extend the upper ontology with application-specific knowledge. By merging upper and domain ontologies at runtime, the environment model can dynamically adapt to any target environment. The current context of use is captured by instances of these ontologies
which are distributed amongst devices for performance and scalability reasons. To accommodate the need of pervasive applications to acquire context and get notified of context changes, we support context retrieval on demand and propagation of context to interested resources on change.

The environment model we proposed in this chapter is used in the heart of several frameworks and tools outlined in chapters §5 §6 §8 and §9. We further extended it with support for behaviour rules that describe how applications should react to sudden context changes, as explained in chapter §5.
Chapter 3

Modelling Reactive Systems

Contents

3.1 Introduction ........................................... 31
3.2 Behaviour rules ........................................... 33
3.3 Behaviour ontology ........................................... 33
  3.3.1 Events, conditions and actions ......................... 34
  3.3.2 Groundings ........................................... 35
3.4 Distributed rule management ............................... 36
  3.4.1 Defining behaviour ..................................... 36
  3.4.2 Explaining, correcting and predicting behaviour ...... 38
3.5 Discussion ............................................. 39

3.1 Introduction

Pervasive applications should be aware of their contexts in order to operate in highly dynamic environments and automatically adapt to changes. Baldauf et al. [Baldauf 07] compare several context-aware systems such as the Context Toolkit [Dey 01], Gaia [Román 02], Hydrogen [Hofer 03] and SOCAM [Gu 04a] in [Baldauf 07]. Each of these systems relies on a context model to adapt the operations of an application to the current location of a user, time and other environment attributes without explicit user intervention. Hence these systems aim at increasing usability and effectiveness by taking environmental context into account. This also means that developers of context-aware applications need to specify how applications must react to changing circum-

Configuring the behaviour of a pervasive application is a complex task as context data may change rapidly and decisions about how to adapt must be made at runtime. Tools not only focus on developers to prototype the behaviour of applications, but also incorporate end-users in this process [Gajos 02]. Moreover, it is important that users can obtain a clear view on the overall system to understand why something (not) happened and hence avoid confusion [Bellotti 01]. This is also true for developers who need to debug their applications at runtime.

With the behaviour model presented in this chapter, we extend the environment model introduced in chapter 2 with support for behaviour rules which specify how to react to a context change. The behaviour model is designed according to the same criteria we envisioned in the design of the environment model: open-ended, dynamic, queryable and distributed. It contributes to an enhanced visibility of the behaviour of pervasive applications at runtime. To this end, tools are indispensable to create and exploit rules in useful ways:

- **Define** behaviour: tools are required to intuitively compose behaviour rules and to edit or remove them afterward. A common approach in interfaces for end-user programming [Gajos 02] is the use of digital or physical building blocks that can be assembled using graphical editors (e.g. by assembling jigsaw pieces [Rodden 04]) or through physical interaction (e.g. using tangible programming parts [Horn 07]). An alternative approach towards defining behaviour in a reactive system is by learning from examples the user provides. In Exemplar [Hartmann 07], patterns are recognized and recorded while interacting with the system: the user demonstrates how to link sensor-based input data with application logic and these examples are then generalized into rules.

- **Explain** behaviour: users should be able to interrogate the environment about sudden changes and get feedback they can understand [Myers 06, Lim 09]. Ko et al. have shown that interactive debugging tools such as the Whyline aid end-users and programmers to better understand and debug the state of an application [Myers 06]. Pervasive applications, however, are driven by context changes and events that occur as a result of these changes. To explain the behaviour of an application, we need to trace the source and cause of an event at runtime which demands for extra information that relates changes in the environment configuration with the resources that caused them to happen. The chain of events and actions is vital information for accurate answers to why (not) questions in an intelligent environment [Lim 09].
3.2 Behaviour rules

- Correct unwanted behaviour: behaviour rules can cause unwanted side-effects when they are executed unexpectedly, for example when gestures are recognized by accident that give rise to events. To recover from mistakes, an undo mechanism is required that assists users in correcting unwanted behaviour.

- Predict behaviour: by querying behaviour rules we can get insight in how the system will respond to certain events. Hence we can interrogate the environment about what might happen when e.g. a button is pressed, before it actually takes place (feedforward).

By modelling an application’s behaviour, we can reuse tools for analyzing and configuring the behaviour of different applications at runtime.

3.2 Behaviour rules

Event-Condition-Action (ECA) rules arise from database theory [Paton 99] and have also been proposed to build reactive systems [Dey 01, Shankar 05]. When an ECA rule is triggered because of an event that occurs, its condition is verified and its action is executed. However, current ECA-based systems are still difficult to understand and manage by end-users. This is largely due to the fact that users can get easily confused by things happening beyond their control [Bellotti 01]. The large number of rules typically required to achieve a useful behaviour also increases the complexity of the overall system. We aim to reduce this complexity and enhance the visibility of a reactive system by linking context information with the behaviour of an application. Hence we can provide more accurate explanations about why things (not) happen. In order to let the system recover from mistakes, we have extended ECA rules with inverse actions, denoted as $ECAA^{-1}$ rules. An inverse action allows users to return to a former state by undoing the rule’s action that caused the unwanted state.

3.3 Behaviour ontology

As an extension to the environment model discussed in chapter 2, we suggest a separate model for capturing the context-driven behaviour of an application. The behaviour model is built up from behaviour rules which specify what should happen when certain context changes occurs. These behaviour rules are described in a behaviour ontology which is depicted in figure 3.1. Similar
to a domain ontology, the behaviour ontology is built on top of the upper environment ontology that is depicted in figure 2.2. In the next section we elaborate on the concepts that populate the behaviour ontology.

Figure 3.1: The behaviour of an application is expressed using $ECAA^{-1}$ rules which are described in a behaviour ontology.

### 3.3.1 Events, conditions and actions

The behaviour ontology defines *events*, *conditions* and (inverse) *actions* as building blocks to construct $ECAA^{-1}$ behaviour rules:

- **an event** is defined as a combination of a Resource and a Sensor instance, i.e. a \{Resource, Sensor\} pair;
- **a condition** is a boolean function which evaluates true (condition is met) or false (condition is not met);
- **an action** modifies the state of one or more resources in the environment.

Consider for example the rule in figure 3.2 which indicates that a light has to be switched on when motion is sensed (e.g. when a user walks by). The event that serves as input for this rule is composed of a hardware resource and its MOTION SENSOR. The rule has no condition and its action (TURNONACTION) switches on a light resource. An inverse action (TURNOFFACTION) can be executed to reverse the effect of the rule, i.e. turn off the light resource again. Actions are further annotated with information about the events they might trigger. For example, the TURNONACTION and TURNOFFACTION might give rise to a \{LIGHT, ONOFFSENSOR\} event. This information is valuable to relate events with actions and thus rules that caused them to happen.
3.3 Behaviour ontology

Figure 3.2: Example of a behaviour rule that turns on a light resource when motion is sensed.

3.3.2 Groundings

Whereas events are fired upon change, conditions and actions are invoked on demand when a rule is triggered. We expect the logic that implements an action to reside in a service or script running at a remote device. Hence, executing an action corresponds to invoking a remote procedure call (RPC). Since different technologies can be used to implement RPC mechanisms (e.g. Java RMI, OWL-S [OWL-S 04]), we will take a similar approach as we did with services (see section 2.3.5) to link models and their underlying technologies: a METHODGROUNDING connects an ACTION instance with its implementation which is illustrated in figure 3.3. Unlike services which provide several methods packed in a software component, an action refers to a single method. A grounding for such a method (e.g. a JRMIGROUNDING) defines inputs and outputs in which the state of resources is passed or returned before they were altered by the action. Rules need to cache this information to make actions reversible: the old state of an updated resource serves as a reference for an inverse action in order to restore the previous situation. Hence the outputs of an action should match the inputs of its inverse action. In a grounding for a condition, the type of a single output is constrained to a boolean datatype.
3.4 Distributed rule management

In this section we describe a reference architecture for the behaviour model. Similar to the environment model, the behaviour model is dispersed over heterogeneous computing devices. Instances of rules are stored in a central rule store (see figure 3.4) which is built on top of the context store (see figure 2.9). Conditions and actions on the other hand, are defined and executed on devices where they can modify the state of local resources. A rule engine listens for incoming events and selects rule instances from the rule store that match the event. For each selected rule, the condition is checked (if any) and if the check passes, the rule’s action is executed. If the rule engine receives an undo request for a rule that was previously executed, the rule’s inverse action is executed. This is achieved by passing the cached state information outputted by the rule’s action to its inverse action. For example, a rule with an action that dims a light to 30% of its maximal intensity should store the previous state of the light in order to allow the light’s intensity to be reversed to its previous state. Clearly, updates to resources that were performed after the action was executed (e.g. by other rules and actions that were executed), will be lost after an undo operation. Note also that a redo operation can be supported, by simply re-executing a rule’s action after its inverse action was executed.

3.4.1 Defining behaviour

To keep users in control of their environment, it is important they can adapt the behaviour of applications to their own preferences, also denoted as end-user programming [Gajos 02]. To effectively configure pervasive applications using a rule-based behaviour model, tools are required to insert behaviour rules in the model and to edit or remove them afterwards. We classify approaches towards creating such rules in three categories, ranging from a low-level approach...
3.4 Distributed rule management

Figure 3.4: Reactive behaviour is stored as a set of rules in a central rule store which runs at the same device ($D_S$) as the context store (see figure 3.4). The rule engine listens for incoming events and selects matching rules. Conditions and actions refer to application logic implemented in a script or service running at a client device ($D_1, D_2, D_3$) and hence are remotely evaluated and executed.

to higher-level approaches:

• **Behaviour scripts:** Scripts can be edited and executed at runtime which makes them very suitable for programming the runtime behavior of applications and thus to add rules to the behaviour model. Although scripts are targeted at (amateur) developers, end-users can still enable or disable them at runtime, giving them limited control over what will (not) happen when the state of the environment changes.

• **Visual programming languages (VPLs):** VPLs mask programming code with visual constructs and help non- or less technical users to program an application. Although VPLs have a broader target public than scripts, it is sometimes more complex to visually model a rule using a generic VPL than to write a few lines of code in a script. Yet, VPLs targeted at a specific user group and domain can enhance the experience of configuring the behaviour of applications. Consider for example the Kodu project[^1] where children can create a game using an icon-based programming interface. The Scratch programming language[^2] also focuses on a young audience to create interactive applications such as stories and games using a graphical interface. In iStuff [Ballagas 03], an intuitive patch panel is used to combine user input and sensor values in a meaningful way.

• **Application-specific configuration interfaces:** Dedicated user interfaces for configuring an application can add or remove behaviour rules in the

[^2]: http://scratch.mit.edu/
background while the user interacts with the interface. Consider for example a configuration interface for a thermostat that allows its users to specify temperature offsets and timings to (de)activate the central heating. An advantage of modelling the thermostat’s configuration by means of behavior rules is the ability to debug the thermostat’s behaviour using generic tools. Moreover, rules provide the option to undo actions or can be disabled. However, high-level user interfaces for configuring an application are limited to pre-defined behavior scenario’s that were considered at design time and thus are less powerful than scripts or VPLs.

3.4.2 Explaining, correcting and predicting behaviour

The state of a pervasive environment can be modified either by end-users interacting with it or by the computing system, programmed to react on certain events. A typical pervasive environment is characterized by a combination of user-driven and system-driven behaviour as users interact with the environment while they are assisted by the computing system:

- **User-driven behaviour**: interacting with an application (e.g. pressing a button in a user interface) triggers an action that can result in context changes which on their turn give rise to new actions and so on. Even though a user is aware of her interaction with a resource, she might still be surprised by the effects of the interaction. This is even more an issue in an environment where multiple users interact at the same time: did I cause this behaviour or someone else and how can we undo it?

- **System-driven behaviour**: when the pervasive computing system is programmed to react on events, it will automatically invoke actions which also might trigger new events. Since events can occur without the user even interacting with the computing environment (e.g. when a sensor value is updated), it can be very confusing to understand why the system behaves in a certain way [Bellotti 01].

In order to provide accurate explanations about why something happened, we must keep track of a history of recent user actions (interaction) and system actions (rules) that were executed. If we consider user interaction as a special type of $ECAA^{-1}$ rule – with a $\{\text{Resource,UserInteractionSensor}\}$ pair as event, no condition, the action that was executed as a result of the user interaction and optionally an inverse action – we can treat user and system actions uniformly. The overhead for a user interface designer to log interactions should be minimal when an API for the behaviour model supports a
3.5 Discussion

The advantage of storing behaviour rules in a semantic model particularly lays in its querying facilities: queries are a powerful instrument to observe the runtime behaviour of applications as they allow to:

- predict what will happen: a query can ask which rules will be executed when a sensor is triggered. By evaluating the conditions of these rules, the pervasive software system can estimate which actions will be executed. Although we are dealing with uncertainty – the state of the environment can change and the rule’s condition can evaluate differently when the event is actually triggered – a better understanding of the pervasive system can be achieved. To make more accurate assumptions, we annotate behaviour rules with extra information about the events they might trigger (see figure 3.2).

- understand why something happened and undo it: if a rule is executed, this is because an event was triggered. The sensor and the resource that caused the event can be traced; a description of the rule’s condition and action helps to understand why the rule was executed, by whom (user or system) and what it did exactly. If an event happens directly after an action is executed that is marked to trigger this type of event, the event is probably related with the action. Moreover, the rule’s action can be reversed by executing its inverse action.

Figure 3.5 depicts a prototype tool for debugging the context-driven behaviour of pervasive applications. The tool is part of the ReWiRe framework (see chapter 5) and relies on information provided by the behaviour model discussed in this chapter. The tool visualizes all the events that take place on a timeline or table view, grouping similar events to avoid cluttering. When selecting an event, the user is presented with an overview of the rules (actions) that were executed as a reaction to this event. The source of the event can also be traced: was it caused by a behaviour rule or by an end-user? Moreover, users can correct unwanted behaviour via an undo option that is supported by the tool.

3.5 Discussion

In this chapter we described a model that adds support for behaviour rules to the environment model introduced in chapter 2. These rules can be used
to build reactive systems that give rise to ambient intelligent environments as rules specify which actions should be executed when events occur in the environment, provided that certain conditions are met. By incorporating inverse actions in behaviour rules, we have illustrated that it is possible to return to a former state, e.g. when behaviour rules were triggered accidentally. Still improvements are needed to guarantee smooth transitions from a new state to an old state. When several behaviour rules are executed at the same time (e.g. because they react to the same type of events), rules might interfere with each other objectives. This could be avoided by modelling conflicts between rules and assert that conflicting rules must not be active at the same time. However, since rules can be provided by different sources that are unaware of the effects of each other’s rules, this will be hard to enforce.

We have also explained how the behaviour model can be leveraged by tools to let end-users configure the behaviour of applications and thus create rules. Moreover, it is important to make users aware of the things that (might) happen in their environment and help them to understand why this is (not) the case. The role of the developer is no longer constrained to delivering pre-configured applications, but also involves the development of end-user tools to improve the configurability and understanding of reactive systems by its end-users. In chapters 5 and 6, the behaviour model and related tools are used to create do-it-yourself applications whose execution can be observed and corrected at runtime.
Part II

Developing and Deploying Pervasive Applications
Chapter 4

Making Resources Talk

Contents

4.1 Introduction .............................................. 43
4.2 W2P framework ........................................... 45
4.3 Messages: a vehicle for data exchange ................. 47
  4.3.1 Addressing peers and groups ..................... 48
  4.3.2 Sending and receiving messages .................. 49
  4.3.3 Monitoring the message flow ...................... 52
4.4 Case study: distributed media player .................. 53
4.5 Discussion ............................................... 55

4.1 Introduction

Several technologies have been proposed to interconnect software services and create a Service-Oriented Architecture (SOA), ranging from distributed object architectures such as CORBA\(^1\) and JXTA\(^2\) to message-oriented middleware infrastructures such as JMS\(^3\) and XMPP\(^4\). Communication middleware has also been proposed specifically for ubiquitous computing systems: MundoCore\(^5\) supports a mix of communication styles and runs on different platforms. Since services in a pervasive environment are diverse in terms of the

\(^{1}\)http://www.corba.org/
\(^{2}\)http://java.sun.com/othertech/jxta/
\(^{3}\)http://java.sun.com/products/jms/
\(^{4}\)http://xmpp.org/
programming languages and toolkits that were used to create them, a scalable network layer is needed to let them exchange information. Our goal in this chapter is to investigate how we can set up ubiquitous communication channels between resources that populate the environment. In particular, HTTP is a suitable candidate to make resources talk over a network because of several reasons:

- it is the Web’s major communication protocol and is supported on any device that runs a Web browser;
- it imposes low requirements on resources and hence can also be used on devices with limited memory and processing power;
- it is understood and spoken by applications running inside a Web browser;
- it does not require special firewall configurations when standard ports 80 or 8080 are used.

The various open standards for Web services such as WS-addressing and WS-eventing are a valuable source of information to get insight in networking requirements such as addressing strategies and message exchange patterns. However, due to their complexity and close relation with SOAP – “SOAP originally stood for Simple Object Access Protocol but tellingly, they’ve dropped that acronym” [Tate 04] – these specifications hardly lead to a light-weight notation. Moreover, toolkits implementing WS specifications mutually diverge in features, architecture and API which is a result of the fact that protocols are standardized, not their implementations [Vinoski 03]. This makes it hard to leverage knowledge from one toolkit to another and implement Web services in a consistent way across platforms. Since Web services depend on a Web server, they are also not particularly suited for deployment on mobile devices either.

Another notable open standard that can be used to interconnect resources is the Extensible Messaging and Presence Protocol (XMPP). While XMPP is mainly useful to develop instant messaging applications, resources can also be considered as chat buddies that communicate by exchanging instant messages. Pierce et al. [Pierce 08] have shown that this instant messaging protocol provides the required features and flexibility to build cross-platform distributed applications. However, XMPP has some shortcomings to make resources talk in a pervasive environment: synchronous data exchange and message attachments are not supported by XMPP, which would be convenient features to
4.2 W2P framework

We propose a framework called Web to Peer (W2P) that combines the simplicity of RESTful communication over HTTP [Richardson 07] with concepts addressed in the WS specifications and the strengths of XMPP’s architecture. Unlike protocols such as UPnP and Jini, W2P does not focus on the discovery of networked entities using multicast, yet a ‘buddy list’ can be retrieved similar to XMPP. When W2P is combined with the environment model presented in chapter 2, resources can be discovered by simply querying the model in which each resource is advertised with a reference.

W2P mimics the architecture of XMPP: a number of W2P clients are interconnected by a W2P server (W2PS) as depicted in figure 4.1. The W2PS fulfills a double role: it acts as a naming service that provides clients with a unique name on the network and routes messages to their destination, similar to an IP router. Once registered with a server, clients can communicate with each other by exchanging messages. W2P clients try to maintain a persistent connection with the W2PS. If for any reason the connection is interrupted (e.g. weak signal strength in a wireless network), a W2P client automatically tries to reconnect. The W2P server regularly polls clients to check if they are still alive. If a client does not respond within a specified time frame, the connection is dropped and the client will need to register with the server again.

The W2PS is designed as a Web application (w2ps.war, 1.2 MB) with open REST architecture. It has been developed using Java servlet technol-
ogy and is deployed on a Web server such as Tomcat or Jetty. A W2P server routes messages to their destination, just like an IP router forwards IP packets. Through this gateway, W2P peers (i.e. entities that are registered on a W2P server using a W2P client) are assigned a unique name on the network. From this point on, peers can exchange messages over HTTP and discover each other via the W2PS’ naming service which maintains a list of registered peers. Both server and clients use message queues to deal with network and message processing delays. The W2PS has a shared message queue for incoming messages and a per-client queue for outgoing messages. The queues implemented in a client resemble incoming and outgoing mailboxes. A software entity can simply put a message in its outgoing queue and short time later it will arrive in the recipient’s incoming queue when the latter is online. Under the hood, the message is sent to the W2PS as soon as a network connection is available. Next, the W2PS routes the message to the recipient’s outgoing queue (at the W2PS). When the recipient is online, the message is transferred to its incoming queue. Otherwise the message is kept at the W2PS and the target entity can pick it up later. As such, clients can process incoming and outgoing messages at their own tempo and engage in asynchronous communication, at least provided that queues are large enough to temporarily cache unprocessed messages.

Since clients do not run a server and depend on a minimal set of libraries, they can easily be embedded in software components, also on mobile interaction devices. For example, the footprint of a cross-platform W2P client library for Java (w2p.jar) is only 45 KB and its dependencies (HttpClient and log4j) measure up to less than 1 MB in total. We also implemented W2P client libraries for C++ and C# to support a wide range of platforms by default. A W2P library for target platforms for which no implementation is readily available, can be created by leveraging W2P’s open REST API. Table 4.1 provides an overview of the main operations supported by this API which are further discussed in the next sections.

http://java.sun.com/products/servlet/
http://tomcat.apache.org/
http://jetty.codehaus.org/jetty/
http://hc.apache.org/httpclient-3.x/
http://logging.apache.org/log4j/
### 4.3 Messages: a vehicle for data exchange

When resources can exchange messages in a similar way as people do in the digital world – instantly over chat or asynchronously using mail or sms – they only have to agree on the language they will use in their messages. W2P does not dictate such language, but rather provides a generic means for software entities to compose, send and receive messages, similar to mail and chat applications. A W2P message carries a number of headers with meta-data (e.g. sender, recipient(s), ...), a payload with the message's content data and possibly a number of attachments. The payload can contain any kind of information: plain text, XML data, binary image files, etc and the content type is specified as a parameter in a message. Next to parameters (key/value pairs that enable fast processing), messages can also contain text or binary attachments. Attachments are particularly useful to separate actual data (e.g. a set of images) from a description of this data in the message body (e.g. an XML document describing the attached images).

To transfer messages over a network, W2P defines an HTTP binding as illustrated in figure 4.2. While SMTP or lower-level protocols such as TCP are also suitable candidates to transport messages, HTTP has the clear advantage that it is supported by Web browsers and most firewalls allow clients to fetch and post HTTP messages without any hindrances. XMPP supports a similar extension to transport messages over HTTP\(^{10}\) Table 4.2 displays the extra HTTP headers that are used by W2P to (de)serialize a message over HTTP. These headers add support for addressing, (a)synchronous message exchange, parameters and attachments.

---

Table 4.1: Main W2P operations.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>register(name)</td>
<td>Register a peer with a W2PS.</td>
</tr>
<tr>
<td>unregister()</td>
<td>Unregister a peer with a W2PS.</td>
</tr>
<tr>
<td>subscribe(group)</td>
<td>Subscribe a peer to a group.</td>
</tr>
<tr>
<td>unsubscribe(group)</td>
<td>Unsubscribe a peer from a group.</td>
</tr>
<tr>
<td>send(message)</td>
<td>Send a message.</td>
</tr>
<tr>
<td>receive()</td>
<td>Receive a message.</td>
</tr>
</tbody>
</table>

Software entities can use a W2P client to register themselves as peers on the network. Peers are addressed by the name they acquire upon registration with a W2P server that acts as (dynamic) naming service. If no self-chosen name is provided, a unique name is generated and assigned by the W2P server. Hence entities can talk with each other using meaningful names instead of IP addresses. In fact, multiple software entities running at the same computer can share an IP address and still have a unique W2P network name. For example, a service that is referred to by a URI (e.g. http://edm.org#MyService) could use this URI as its network address, making communication as transparent as possible. This proofs to be a simple but effective approach in practice. Distributed applications often depend on a predefined set of software entities that live during the application’s lifetime. When these entities acquire a static
name, other entities can directly access them without the need to discover them first. Entities that simply join the network to interact with a particular service are not directly addressed by others and hence can be assigned a randomly generated name.

Apart from addressing peers individually, peers can join and address groups. Groups resemble mailing lists: they allow to send a message to multiple recipients at once, without having to know who is actually subscribed to the list (group). Groups are hierarchically organized in a tree structure: a root group ([]) has a number of child groups (e.g. [games], [devices], ...) which can have their own child groups (e.g. [games][pacman]) and so on. Messages sent to a group are forwarded to the peers in this group and the members of its ancestor groups. For example, a message sent to [games][pacman] is sent to all peers subscribed to either [games][pacman], [games] or []. Groups are particularly useful for:

- addressing a group of similar resources at once, e.g. [users] or [games];

- subscribing to individual events (e.g. [pacman][events][killed]) or a range of events (e.g. [pacman][events]).

### 4.3.2 Sending and receiving messages

Asynchronous and synchronous communication are common needs for a distribution application [Coulouris 05]. Although most of the communication in a pervasive computing environment will be asynchronous, there are also situations where an application must explicitly wait for a reply message before it can continue its execution, for example when it needs to execute a remote query and depends on its results. Therefore W2P offers asynchronous and synchronous message exchange patterns together with a mechanism to dispatch incoming messages to dedicated message handlers. We support the following exchange patterns through dedicated ‘send’ methods in a W2P client which are further clarified in figure 4.3.

- **Asynchronous one-way**: a message is sent, the call returns immediately and no reply message is expected. This is suitable for event messages.

- **Asynchronous two-way**: a message is sent, the call returns immediately and a reply message is expected which is passed to a message handler in a separate thread as soon as it is received.
• **Synchronous**: a message is sent and the call blocks until a reply message is received or a timeout occurs. This is useful to execute a remote procedure and immediately return its results.

![Figure 4.3: Messages can be exchanged using different patterns.](image)

Messages sent using W2P are guaranteed to arrive at their destination in the same order as they leave a W2P peer and they will be delivered only once. However, delivery of a message is not assured when the asynchronous one-way exchange pattern is used. For instance, message will be dropped when a receiver gets disconnected and unregistered from the W2PS (e.g. due to a crash). In the case of asynchronous two-way and synchronous message exchange, a reply message confirms that the original message was delivered and a timeout error notifies peers of a delivery failure. For most applications these exchange patterns will be sufficient, yet applications with special demands can implement custom exchange patterns on top of W2P. For example, delivery of a response message can be confirmed too by sending a dedicated reply (delivery notification message). Since the chain of messages in a conversation between peers is linked together by message identifiers, it is easy to differentiate between reply messages and messages that are part of other conversations.

A rule-based dispatch mechanism that is built into W2P automatically passes incoming messages to dedicated message handlers. An incoming message is matched against a set of dispatch rules that relate a number of message properties (e.g. sender, recipient(s), type of the message, . . .) with a message handler. When an incoming messages matches a rule, the message is passed
4.3 Messages: a vehicle for data exchange

as an argument to the handler defined in the rule. This mechanism is illustrated in listing [4.1] It shows that a message of type `MyRequest.TYPE_NAME` will be delegated to a `MyRequestHandler` instance, independent of sender and receiver fields in the message's header (indicated by asterisks) and that an incoming message of type `MyEvent.TYPE_NAME` addressed to 'MyService' will be delegated to a `MyEventHandler` instance.

```java
class MyProgram {
    private Peer m_peer;
    private Dispatcher mDisp;

    public MyProgram() {
        m_peer = new Peer('MyService');
        mDisp = new Dispatcher(peer);
        MyRequestHandler rh = new MyRequestHandler();
        MyEventHandler eh = new MyEventHandler();
        mDisp.addRule(new DispatchRule("*", "*", MyRequest.TYPE_NAME, rh));
        mDisp.addRule(new DispatchRule("MyService", "*", MyEvent.TYPE_NAME, eh));
    }
}

class MyRequestHandler extends Handler {
    public void handle(Message m) {
        MyRequest req = new MyRequest(m);
        // process request and send response
        MyResponse resp = new MyResponse();
        Message rm = resp.toMessage(getPeer().getName(), m.getFrom(), m.getId());
        peer.send(rm);
    }
}

class MyEventHandler extends Handler {
    public void handle(Message m) {
        MyEvent evt = new MyEvent(m);
        // do something
    }
}
```

Listing 4.1: Dispatching incoming messages via rules to dedicated handlers.
To abstract from raw messages, W2P introduces a `Type` class which represents an object in an application that can be (de)serialized into a W2P message. For example, the `MyRequest` and `MyEvent` objects in the source code are inherited W2P types. A data binding layer assists developers with the conversion between messages and native objects. W2P includes classes to simplify data binding across platforms when XML is used to structure a message’s payload. Using XPath [XPath 07] expressions data can be selected from a message and automatically converted to a preferred datatype (string, integer, array of floats, ...) as illustrated in listing 4.2.

```java
class PlayRequest extends Type {
    int m_track;

    public void fromMessage(Message m) {
        String xml = new String(m.getBytes());
        XmlBinder binder = new XmlBinder(xml);
        m_track = binder.getInteger("/PlayRequest/track/text()");
    }
}
```

Listing 4.2: Converting a W2P message to a native object.

When elevated with efficient parser techniques such as presented in XML Screamer [Kostoulas 06], the data binding layer can reduce the overhead inherent to data (de)serialization to a minimum for the developer. Plenty of other toolkits can also be used in conjunction with W2P to automate the mapping between message data and native objects (e.g. JAXB 11 or Liquid XML 12 when dealing with an XML-based payload).

### 4.3.3 Monitoring the message flow

The W2P framework includes a Web interface targeted at developers that provides an overview of the currently registered peers on a W2P server and the groups they are subscribed to. This information is useful to understand why messages are (not) delivered at specific peers. Moreover, by capturing messages and presenting them to developers or even end-users when visualized appropriately, the flow of context information between resources can be

---

monitored. The debug interface presents all the messages exchanged over the network in a certain time span, provided that a W2P server runs in promiscuous mode (an option that can be disabled for security reasons). Figure 4.4 shows a W2P message with attachment in the Web interface.

![Message Flow in a W2P Network](image)

Figure 4.4: The message flow in a W2P network can be monitored in a Web browser. A Web interface provides an overview of registered peers and the groups they are subscribed to and can be used to inspect messages.

### 4.4 Case study: distributed media player

As a proof of concept, we W2P to connect a distributed user interface with a media player (see figure 4.5(a)). We built a C++ plugin for the open-source XMMS media player to make the media player’s main functions available to W2P peers over the network. We also designed a user interface running at a PDA to remotely operate XMMS, created using Windows Forms in C#. Due to the scalability of W2P clients, the diversity in platforms and development technology is not an issue. On the contrary, it promotes the design of a distributed system where technology matches a device’s software platform.

Figure 4.5(b) illustrates the sequence of actions that take place when the ‘play’ button is pressed in the user interface and listing 4.3 depicts code fragments that are executed in the XMMS plugin. It shows that both the XMMS plugin as the user interface component first register with the W2PS and that the latter subscribes to player events which are addressed to the [xmms][events] group. From this point on, they can exchange messages. To the developer, it appears as if native objects (PlayRequest, PlayerEvent) are ‘emailed’ over the network. These objects are actually transferred as messages over HTTP and (de)serialized by a W2P client.

(a) XMMS can be remotely operated via a migratable user interface. (b) User actions and player events are exchanged in W2P messages.

Figure 4.5: A distributed user interface on a PDA is interconnected with a media player running at a desktop PC through W2P.

XmmsPlugin::XmmsPlugin()
{
    m_peer = new Peer("xmms");
    m_disp = new Dispatcher(m_peer);
    PlayRequestHandler *h = new PlayRequestHandler();
    m_disp->addRule(new DispatchRule("*", ",", PlayRequest::TYPE_NAME, h));
}

PlayRequestHandler::handle(const Message *m)
{

4.5 Discussion

In this chapter we presented a message-oriented middleware to interconnect resources that inhabit a pervasive environment over a network. The W2P framework provides a light-weight solution to exchange messages over HTTP between different platforms, either asynchronously or synchronously. As W2P entities can choose their own network name (e.g. a URL) and look up other registered entities, they can be addressed transparently without prior knowledge of IP addresses. This is particularly useful in dynamic environments where resources are not known in advance. Furthermore, a hierarchical publish, subscribe and dispatch mechanism accommodates message filtering and delegation of incoming messages to dedicated handlers.

Since W2P clients do not run a server and generate default Web traffic, W2P can operate behind firewalls and runs on most internet-enabled mobile devices. We believe a merit of W2P technology is the ease of turning new or existant functionality into a Web-accessible service and this in a consistent way across platforms. In a case study we illustrated how a media player can be operated via a distributed user interface using W2P. W2P is also used as communication middleware in the ReWiRe framework discussed in chapter 5.

Listing 4.3: Main methods that are executed to serve a play request in the XMMS plugin.
5.1 Introduction

Most software systems for building pervasive applications still expect a static environment where the devices and services that are used to complete a task and the number of users that interact with an application do not change in one usage session. When the environment is allowed to change while users are interacting with it, this also means users will be able to interact in different ways with the environment that can only be determined while the software
ReWiRe: Middleware for Pervasive Applications

System is in use. This situation is tackled in [Peters 03] and [Chen 04a] where a semantic layer for describing the context of use is exploited to support context-aware pervasive systems. Most attention in this work goes into supporting changes in an existing architecture, although dynamically composed. We seek to support applications that have to cope with a changing configuration during usage and where the pervasive computing system as well as the end-user can coordinate the effect of these changes. Grimm et al. [Grimm 04] identify three requirements system support for pervasive systems must meet:

- System support needs to embrace contextual change and not hide it from applications.

- System support needs to encourage ad hoc composition and not assume a static computing environment with a limited number of interactions.

- System support needs to recognize sharing as the default.

The environment model discussed in chapter 2 is an important facet to fulfill these requirements, as it can describe the semantics of a pervasive environment, dynamically integrate newly discovered resources and supports sharing of resources across devices. We use it as a foundation in a middleware infrastructure called ReWiRe that provides support for the development and deployment of pervasive applications. With ReWiRe we propose a platform for pervasive applications that can adapt itself when new configurations arise such as the usage of a new mobile device. Our approach links the environment configuration with the software architecture, denoted as a dual world process. On the one hand, an environment model describes the context of use and on the other hand, the context of use is reflected in the deployment of service and user interface components. This link between model and architecture is continuously maintained during the life time of a pervasive application. When the environment configuration changes, the software architecture changes accordingly. We refer to this dynamic (re)configuration process with the term ‘(re)wiring’. Achieving collaborations by bringing together several users and devices is part of a (re)wiring process.

In this chapter we explain how ReWiRe can be used to develop different components that are part of a context-aware pervasive application. Chapter 6 further elaborates on a design strategy (called PerCraft) for this type of applications that exploits ReWiRe’s (re)wiring capabilities. In the next section we first introduce the OSGi Service Platform since our ReWiRe software is built on top of it.
5.2 Open Services Gateway Initiative

The Open Services Gateway Initiative (OSGi), also known as the Dynamic Module System for Java, defines an architecture for modular application development. OSGi technology provides a service-oriented, component-based environment for developers and offers standardized ways to manage the software life cycle. These capabilities greatly increase the value of a wide range of computers and devices that use the Java platform.

Similar to a Web container in which Web applications can be deployed, an OSGi container such as Apache Felix[^1][^2][^3] Knopflerfish[^2] and Equinox[^3] manages the life cycle of OSGi modules, also referred to as ‘bundles’. These OSGi bundles can be remotely installed, uninstalled, updated, started and stopped without the need to restart the container. Each bundle includes a manifest (MANIFEST.MF) that configures the bundle and defines its imports (dependencies) and exports (what it offers) as illustrated in the example manifest in listing 5.1.

[^1]: http://felix.apache.org/
[^2]: http://www.knopflerfish.org/
[^3]: http://www.eclipse.org/equinox/

```
Bundle-Name: Hello world
Bundle-SymbolicName: org.helloworld
Bundle-Description: Hello World bundle
Bundle-Version: 1.0.0
Bundle-Activator: org.helloworld.Activator
Bundle-ClassPath: .
Import-Package: org.osgi.framework
Export-Package: org.helloworld
```

Listing 5.1: OSGi manifest.

An Activator, referred to by the Bundle-Activator header in the manifest, acts as a ‘main’ class whose start and stop methods are executed when the bundle is started or stopped. Listing 5.2 shows an example.

```
public class Activator implements BundleActivator {
    public static BundleContext m_bc;

    public void start(BundleContext bc) throws Exception {
```
```java
System.out.println("Starting: Hello world");
m_bc = bc;
}

public void stop(BundleContext bc) throws Exception {
    System.out.println("Stopping: Goodbye Cruel World");
m_bc = null;
}
}
```

Listing 5.2: OSGi Activator class.

Figure 5.1(a) depicts the different layers in the architecture of OSGi:

- **Bundles**: Bundles are components made by developers.
- **Services**: The services layer connects bundles in a dynamic way by offering a publish-find-bind model for plain old Java objects.
- **Life cycle**: The API to install, start, stop, update, and uninstall bundles.
- **Modules**: The layer that defines how a bundle can import and export code.
- **Security**: The layer that handles the security aspects.
- **Execution Environment**: Defines what methods and classes are available in a specific platform.

![OSGi layered architecture](http://www.osgi.org/)

(a) Extracted from [Ahn 07].

(a) Extracted from http://www.osgi.org/.

(b) Extracted from [Ahn 07].

Figure 5.1: OSGi layered architecture (a) and supported bundle states (b).
5.3 ReWiRe framework

Figure 5.1(b) provides an overview of the different states that are assigned to a bundle in the life cycle layer. The **RESOLVED** state indicates that all Java classes needed by the bundle are available and that the bundle is either ready to be started or has stopped. For a more detailed overview of OSGi and its features, we refer to the OSGi Service Platform specification [Alliance 09].

5.3 ReWiRe framework

ReWiRe relies on the OSGi framework to deploy pervasive software components and use the Apache Felix implementation due to its small footprint and good performance, also on mobile devices. Previous work discusses the use of OSGi technology in a pervasive environment [Lee 03, S. Rellermeyer 07]. Lee et al. [Lee 03] explain how smart spaces can benefit from a standard execution environment such as defined by OSGi. In contrast with [S. Rellermeyer 07] we do not extend the OSGi framework itself to add distribution capabilities, but add semantics separate from the framework that describe how to interoperate with OSGi components or other software services. Figure 5.2 describes the architecture of ReWiRe. We distinguish a server side (host) and a client side that differ in terms of deployed OSGi components:

- A **ReWiRe host** ‘serves’ models and an execution environment for pervasive applications that can be discovered by ReWiRe clients.

- A **ReWiRe client** is installed on each device that participates in the pervasive environment. Services, tasks and user interfaces are deployed on a client, depending on the role of the device it runs on. For example, embedded devices without screen will typically serve interactive resources by means of services and tasks, whereas mobile devices such as a smart phone are suited as interaction device and can present a user interface for a task.

The figure provides an overview of the default layers (OSGi bundles) that make up a ReWiRe host and client:

**Logging** : This layer adds logging capabilities to ReWiRe.

**SLP** : The Service Location Protocol (SLP) is used by ReWiRe clients to discover a ReWiRe host. SLP is a simple service discovery protocol that allows computers and other devices to find services in a local area network without prior configuration. A ReWiRe host announces itself as a SLP service on the local network via a URL such as `service:rewire:`.
ReWiRe architecture.

http://pc-geert:8070 by which it can be located. ReWiRe clients search for a ‘rewire’ service by multicasting a query over the network which is answered by a ReWiRe host, allowing the former to fetch its attributes (i.e. configuration properties) that allow a client to connect with a discovered host).

**HTTP** : The ReWiRe host embeds the Jetty[^4] HTTP server that is used to deploy Web applications. At ReWiRe clients, the layer adds support for accessing resources via HTTP. Jetty was chosen because it has a small footprint for a Web server and it can easily be embedded in other applications, in particular an OSGi component.

**W2P** : Web to Peer (W2P), described in chapter[^4] is used as communication middleware in upper layers. Clients and host rely on W2P peers to exchange context information and answer distributed queries. W2P is also used to share services (i.e. make their API available on the network) and remotely interact with them.

**Jena** : This layer integrates the Jena[^5] semantic Web framework in ReWiRe. At the host, Jena repositories are maintained and clients can remotely

[^4]: http://jetty.codehaus.org/jetty/
[^5]: http://jena.sourceforge.net/
access these repositories through an API. Jena was selected due to its support for OWL and SPARQL and its bindings with reasoners such as Pellet [Sirin 07].

**Environment** : The environment layer leverages the Jena layer to implement the environment model that was discussed in chapter 2. At the host it manages a Jena repository in which the semantics of the environment are stored, in line with the reference implementation of a distributed context model as outlined in section 2.4. Jena’s built-in query engine is used to query this repository and clients expose an API for importing domain ontologies and retrieving and updating information about resources.

**Behaviour** : Similar to the environment layer, the behaviour layer provides an implementation of the behaviour model discussed in chapter 3. At the host it maintains a separate repository for storing behaviour rules and a rule engine that is responsible for selecting rules when events take place and managing the execution of these rules as explained in section 3.4. An API allows clients to populate the behaviour model with rules and a JavaScript engine is used by clients to execute actions and conditions of scripted behaviour rules. We use the Rhino JavaScript implementation since it can leverage existing Java packages and classes. As such, behaviour scripts can access other layers (e.g. the environment and behaviour layers) through Java APIs. This renders scripts as powerful as native Java classes inside an OSGi bundle.

**Client** : The client layer adds functionality to share resources in a pervasive environment. In particular, it allows users, the device(s) they are using, their services, tasks and user interfaces as well as application-specific resources to be advertised in the environment model.

**Client UI** : The client UI embeds a user interface for a ReWiRe client. This user interface resembles a messenger application (see figure 5.3(a) and (b)), looking familiar to most users. After signing in to a ReWiRe host (i.e. registering a user and device in a pervasive environment), the user is presented with a list of available users and devices in this environment and a number of tools which are discussed in section 5.6. Besides, available resources and their supported tasks are presented in a menu and a user interface can be requested for a task as illustrated in figure 5.3(c) and (d). By dragging a task on a user or device, tasks and

their user interfaces can be distributed amongst ReWiRe clients which is elaborated on in section 8.6.

![Login screen](image1)

![Main screen](image2)

![Resources screen](image3)

![Tasks screen](image4)

Figure 5.3: ReWiRe client user interface.

With these default layers in place, ReWiRe provides an implementation of the environment and behaviour model and their distributed architecture that is discussed in chapters 2 and 3. An API for the environment model
5.3 ReWiRe framework

provides developers with Resource, User, Device, Service, Task, Sensor, Aggregator, ... classes that can be used to create and interact with concepts defined in the upper environment ontology, depicted in figure 2.2. The hierarchy of classes as defined in this ontology is reflected in the API (i.e. Device is a subclass of Resource) and class properties can be accessed using get and set methods. Likewise, an API for the behaviour model provides classes and methods to create and manage behaviour rules. As such, the ontologies that underpin the models that capture context and rules in an environment, are abstracted for developers of applications created using ReWiRe. The code fragment in listing 5.3 shows how the state of a resource created and shared on a ReWiRe client can be retrieved from any ReWiRe client connected to the same ReWiRe host.

// ReWiRe client @ device 1: create and share a resource
Device d = new Device("http://edm.org#MyLaptop");
d.setName("Geert's laptop");
d.setIP(InetAddress.getLocalHost().getHostAddress());
EnvironmentClientBundle.getClientService().shareResource(d);

// ReWiRe client @ device 2: get a resource
Device d = new Device("http://edm.org#MyLaptop", true);
System.out.println("Name: " + d.getName());
System.out.println("IP: " + d.getIP());

Listing 5.3: Sharing and accessing distributed resources on a ReWiRe client.

The true flag in the constructor of a Resource class (Device in the example) indicates that the context of a resource with specified URI needs to be resolved. To fetch this data, a SPARQL query like the one in listing 5.4 is composed that asks a ReWiRe host for all properties related to this resource.

CONSTRUCT { <http://edm.org/environment#MyResource> ?p ?o }
WHERE { <http://edm.org/environment#MyResource> ?p ?o }

Listing 5.4: Fetching the properties of a resource.

This query acts as an aggregator for the resource’s data and is evaluated as outlined in figure 2.13: the ReWiRe host runs at device $D_S$ while ReWiRe clients are installed on devices $D_1$ and $D_2$. First, a reference to the resource is retrieved from the context store at a ReWiRe host that depicts at which
ReWiRe: Middleware for Pervasive Applications

ReWiRe client the actual data object that represents the resource resides. This client is then requested to return the current state of the resource as a set of OWL/RDF triples. Finally, the collected data that describes the resource is parsed and made available in a Java object at the requesting client. In the code example, a Device object is used to read out the properties of the resource. The Device class will invokes the constructor of its parent class (Resource) to deserialize generic properties (i.e. name and icon) and then deserializes its specific properties such as an IP address and a list of services.

To conclude ReWiRe’s architecture overview, we summarize its highlights:

- Each device part of the pervasive environment runs a ReWiRe client.
- After a user and device are signed in to an environment (discovered by a ReWiRe client and served by a ReWiRe host) the user and device become available and the user can start interacting with available resources via the tasks these resources support. When a task is executed, a suitable user interface for the task is selected and presented to the user.
- Reading operations on shared resources are supported by default on remote devices (i.e. all devices that run a ReWiRe client other than the one that manages the shared resource). To change the state of a resource from a remote device, a service for that resource is required which exposes an API to operate the resource from remote devices.
- An integrated environment and behaviour model can be queried using SPARQL on a ReWiRe client.
- Behaviour rules are stored on a a ReWiRe host and executed by a script engine on a ReWiRe client for security reasons.

For more information on ReWiRe’s embedded models we refer to part I. In the next section, we show how pervasive applications can be created and deployed using ReWiRe.

5.4 Creating pervasive applications using ReWiRe

A pervasive application built using ReWiRe consists of several dynamic software components (OSGi bundles) which can be deployed on and distributed amongst ReWiRe clients at runtime. Each of these components leverages ReWiRe’s built-in environment model to query the current context of use and
5.4 Creating pervasive applications using ReWiRe

to notify other components of changes in the environment configuration so that
they can adapt accordingly. Creating a pervasive application with ReWiRe
involves two steps:

1. Create an application ontology.

2. Create services and user interfaces that can be dynamically deployed on
a ReWiRe client.

We elaborate on these steps in the next sections.

5.4.1 Creating an application ontology

In an application ontology, the concepts related to an application’s domain are
described. These include different types of resources, sensors and aggregators.
Figure 5.4.1 shows an ontology for an application to operate lights which is
created using the Protégé tool. The rationale behind the concepts in the on-
tology is the following: a number of LIGHT resources populate the environment
and are managed by a LIGHTSERVICE that provides functionality to operate a
LIGHT resource. A graphical user interface (TOGGLELIGHTSWINGUI) allows
end-users to toggle the state of an arbitrary LIGHT resource (TOGGLELIGHT-
TASK).

To use these concepts in a ReWiRe application, they are mapped on Java
classes by a developer using ReWiRe’s environment API. This is achieved by
extending classes in the environment API with information about a concept’s
URI, (de)serialization code and convenience methods. For example, a Light
class extends the Resource class with a few lines of code to (de)serialize an
OWL/RDF representation of a light and methods such as isOn and isOff
make a light’s properties easy accessible to a software developer. Services
(e.g. LightService) extend a Service class and are provided with an imple-
mentation as discussed in section 5.4.2. Tasks and user interfaces are inherited
from Task and UI classes and correspond to instances in the application ontol-
ogy. For example, the ToggleLightSwingUI class corresponds to a SwingUI in-
stance with http://edm.org/environment/light#ToggleLightSwingUI as
URI. A user interface component is added later as outlined in section 5.4.3.
Likewise, sensors and aggregators extend Sensor and Aggregator classes.
Sensors are triggered by a service when it updates the state of a resource
(usually a software resource) or when it senses a state change in a resource
(usually a hardware resource). Aggregators abstract underlying queries – a

http://protege.stanford.edu/
Figure 5.4: An application ontology created in Protégé is mapped on an API (i.e. a set of Java classes) so that the ontology’s concepts can be used efficiently in program code.

query collects context properties – and can be executed on demand. The process of transforming a domain ontology into an API could be automated via code generation.

An application ontology is published by an application provider on a public website so that software components depending on it can easily download the ontology and check for updates. A software component can import this ontology, similar to (dynamically) importing required packages in a Java program. By importing an ontology its data is added to the context store on a ReWiRe host. Once the concepts of an application ontology are aggregated with the environment ontology, corresponding Java classes can be used to create and share resource instances. These resources can then be queried using SPARQL as illustrated in the code example in listing 5.5.

```java
EnvironmentService envs = EnvironmentBundle.getService();
EnvironmentClientService envcs = EnvironmentClientBundle.getService();
// import application ontology
envs.importDomain("http://edm.org/environment/light");
// create and share a light resource
Light l = new Light("http://edm.org/environment/light#KitchenLight");
envcs.shareResource(l);
```
5.4 Creating pervasive applications using ReWiRe

Listing 5.5: An application ontology is dynamically imported at a ReWiRe client. Resource objects corresponding to concepts in this ontology can be then be created and queried using SPARQL.

After developing services and user interface components, the application ontology should be extended with groundings to relate corresponding concepts (e.g. LightService and ToggleLightSwingUI) with the newly created software components. These groundings can also be aggregated at runtime with the application ontology. As such, a service can advertise itself on a ReWiRe host together with a grounding that specifies how a ReWiRe client can access and interact with the service.

5.4.2 Creating services

Services are software components created by developers that add functionality to resources and/or use and produce context information. For example, a service can be created to read out data from a GPS chip and share this information with resources that are interested in the location of a device. Besides, a service installed on a gateway PC connected to electronic circuits in an environment (domotics), can provide functions to control light resources, a thermostat or an alarm system. To share a service and make its functions available to other services, both an implementation and a proxy component need to be developed:

- A service implementation component contains the logic of a service, i.e. an implementation of its functions. To allow sharing, the implementation component should expose an open API and listen for incoming requests to invoke its methods.

- A service proxy component contains logic to remotely interact with a service using the service's URI as a reference to its implementation.

Like any resource in the environment, a service is represented by a URI that is generated at runtime or provided in an application’s configuration file. The code fragment in listing 5.6 illustrates how a LightService can be interacted with using a proxy component.
// ReWiRe client @ device 1: create and share a light service
String uri = "http://edm.org/environment/light#MyLightService";
LightService s = new LightServiceImpl(uri);
EnvironmentClientService envcs = EnvironmentClientBundle.getService();
envcs.shareResource(s);

// ReWiRe client @ device 2: interact with the service through a proxy
String uri = "http://edm.org/environment/light#MyLightService";
LightService s = new LightServiceProxy(uri);
for (Light l : s.getLights())
    s.turnOff(l);

Listing 5.6: A service shared on one device can be interacted with from another
device using a proxy component.

Figure 5.4.2 illustrates the architecture of a LightService. An interface
describing the service’s API (LightService) is inherited and implemented by a
LightServiceImpl and LightServiceProxy class which each end up in a dif-
f erent Java package and corresponding OSGi bundle. In the former class the
application logic behind the service’s API is encoded whereas the latter class
is programmed to access this logic. In this example, W2P is used to intercon-
nect implementation and proxy classes over a network, but other middleware
frameworks such as Java RMI could be used as well. The TurnOnRequest and
TurnOffRequest classes are W2P classes representing the messages that are
exchanged to remotely invoke a method of a LightService instance. When
a service provides its URI as network name to a W2P peer in its implementa-
tion, a proxy for a LightService can send a TurnOnRequest to an instance
of a LightService using its URI as address.

Middleware classes, along with classes that represent concepts in the
application ontology such as LIGHT and LIGHTSERVICE are common classes that
can be shared between implementation and proxy components. Therefore cre-
ating a proxy component should not result in too much overhead, since most
classes that are needed to implement a service can be shared anyway. More
effort is needed though when proxy components are required for various tar-
get platforms. This requires a port of a proxy component to one that can be
deployed on the target platform. In fact, when a service’s API is described
in detail (e.g. using OWL-S), proxy components could be generated automa-
tically from this description, similar to generating a proxy component for a
Web service from a WSDL document. To better support multiple platforms
5.4 Creating pervasive applications using ReWiRe

and decrease the work of a developer, code generation algorithms would be a welcome extension to ReWiRe.

5.4.3 Creating user interfaces

User interfaces allow end-users to interact with a service and to observe its state. When designed manually, the designer can guarantee an acceptable aesthetic quality and make effective use of the input capabilities of the target device. Still different versions of a user interface are required, depending on supported platforms and toolkits on specific target devices which is o.a. tackled by the Jelly tool [Meskens 10]. On the other hand, when user interfaces are (semi-)automatically generated from an (abstract) description, an interface can dynamically adapt to the target device [Gajos 04, Nichols 06]. However, since its concrete layout is decided on at runtime, the quality of a generated user interface highly depends on the renderer that is used.

In the examples and case studies outlined throughout this text, we make use of migratable handcrafted user interfaces and in particular Java Swing and Web interfaces. Since the focus of this work is not on the generation of user interfaces, we prefer visually appealing interfaces that can be easily deployed over generated interfaces which require additional renderers. However, thanks
to groundings that tell a ReWiRe client how to render a user interface (see also section 5.5), any type of user interface can be supported. An advantage of a Java Swing interface is that it can be embedded in the ReWiRe client interface as depicted in figure 5.3(d) and connect with application logic (i.e. distributed services) using readily available Java APIs. The increasing support for modern Web browsers on mobile devices also promotes the use of Web interfaces. In chapter 7 we present a toolkit that brings semantics to the Web browser, allowing Web interfaces to make use of an ontology-based context model as well.

Figure 5.4.3 shows the architecture of a Swing user interface that presents the ToggleLightTask. The interface can discover available LightService instances by querying the environment model and connect with one using the proxy bundle and the service’s URI. However, when multiple service instances are available and a user interface component can not automatically decide which service instance to allocate, user input is required. For example, when a media service is deployed (and shared) on several devices in the environment, a user interface presenting a PlayMediaTask can ask the user which service to operate. In this case, we use the device that runs the service to make selection easier for end-users as shown in figure 5.7.

When a service bundle (implementation) is deployed and started, the service is started as well. A user interface, however, should not pop up immediately when its bundle is deployed. Instead, a factory class is registered on a
5.5 From task to user interface

Figure 5.7: A user interface for a PLAYMEDIATASK can discover and operate available MEDIASERVICE instances of which one can be selected by users using an abstraction more familiar to an end-user, i.e. the device a service runs on.

ReWiRe client that generates a user interface instance upon request. A service can also use an HTTP layer (see figure 5.2) to publish a Web interface on a device. The site URL is then advertised in a grounding for the Web interface.

5.5 From task to user interface

From section 5.4 it should be clear that a ReWiRe application is made up of an application ontology and a set of services and user interfaces. The application ontology acts as a schema describing concepts relevant to the application domain, a set of tasks that can be executed and groundings that specify how services and user interfaces can be deployed on a ReWiRe client. Since an application provider can not take into account all services and user interfaces that can be used in combination with an application, we allow an application ontology to grow at runtime with additional tasks and groundings. This allows a (third party) software component to seamlessly integrate with an application by advertising the extra tasks it adds to the application, along with grounding information that is needed to support these tasks.

Figure 5.8 provides an overview of the steps that are traversed by a ReWiRe client when a user selects a task to execute. It shows that a user interface for
the task is searched for and selected from the context store, possibly assisted by the user if different candidate user interfaces are found. By analyzing the grounding of the selected user interface, a ReWiRe client becomes familiar with the technology that has been used to create the user interface component. Still different types of user interface can be grounded in the same way. For example, a SwingUI and SpeechUI can be packed both in an OSGi bundle; the former providing a graphical interface and the latter serving VoiceXML documents. Hence the combination of a user interface type and its grounding determines how to render a user interface component, e.g. display it on a screen or pass it to a speech engine.

By querying the environment for services a task depends on, a ReWiRe client can determine in advance whether a task can be executed, i.e. if all required service instances are currently present in the environment. If not, it makes no sense to render a user interface for the task since its service proxies will not be able to connect to a service.

### 5.6 ReWiRe tools

Several runtime tools have been integrated in ReWiRe that can be accessed via buttons in the ReWiRe client user interface, as shown in figure 5.3(b). These
tools are deployed on a ReWiRe client via OSGi bundles, similar to service and user interface components. Figure 5.9 depicts the standard tools that have been developed for ReWiRe. A package manager (figure 5.9(a)) eases the deployment of OSGi bundles and shows their state at runtime, which is very useful to verify that groundings were properly interpreted and proxy and user interface components are properly installed and started. A query tool (figure 5.9(b)) allows developers to inspect the environment and behaviour models at runtime using SPARQL queries. In an early version of ReWiRe, these models were visualized as an interactive graph. However, with more and more concepts populating the environment model this visualization soon became too complex, so we decided to discontinue its development in favour of a query interface. In section 3.4.2 we introduced another debug tool that keeps track of the events that happen and the behaviour rules that are triggered as a result of these events. This tools helps to explain the effects of both user-driven behaviour – user actions logged by user interface components using an API in the behaviour layer – and system-driven behaviour, specified by behaviour rules. Such rules can be injected in the behaviour model via scripts that are managed and created by the tools in figure 5.9(c) and (d).

Additional tools have been created in projects where ReWiRe is used as pervasive middleware framework. In [Aksenov 09] a spatial tool is used to simulate movement and rotation of resources which causes spatial relations to be updated in the environment model. In [Vermeulen 10] a why menu is proposed to let users pose questions about things that happen in their surroundings. The why menu is integrated in ReWiRe as a runtime tool and relies on ReWiRe’s behaviour model to explain events that occur in the environment. It provides a user-oriented alternative for the debug tool shown in figure 3.5 which is primarily focused on developers who need to monitor the behaviour of their context-aware applications.

5.7 Case studies

In this section we outline two case studies about building pervasive applications using ReWiRe. The first case study describes a legacy open-source game application that was extended for use in a pervasive environment. In a second case study we used behaviour scripts to connect sensor values of a dedicated input device with several applications that were not designed for it at first.
Figure 5.9: Runtime tools supported by a ReWiRe client.

5.7.1 Tux Racer

Tux Racer is an open source arcade game. In the game, the player controls Tux as he slides down a course of snow and ice collecting herring. The player can choose to control Tux either by keyboard or joystick by default. Various
forks of the game exist and we bundled one of them, Extreme Tux Racer, with ReWiRe to create a ubiquitous game experience. Unlike the original game, our pervasive variant scales to an heterogeneous environment and encourages collaboration by allowing gamers to compete each other in a race using the devices at hand. As test environment, we used a room that is equipped with a tabletop display with integrated PC, a PC with projector and a set of PDAs. Both PCs run the default ReWiRe client and on the PDAs we installed a light-weight C# version of the ReWiRe client software, allowing users to register themselves in the environment and configure a game instance using a distributed user interface. Opposed to ReWiRe for Java, the C# clone does not support the deployment of OSGi bundles but can load native libraries instead. Tux Racer user interfaces are encapsulated in such a library and advertised in the environment model using a custom grounding. An instance of an application ontology for ubiquitous Tux Racer depicts the setup of the test environment in figure 5.10. The tabletop and projector PCs run a TuxRacerService which relies on a slightly modified version of the Extreme Tux Racer application. The original C++ source code has been extended with a W2P API so that a game instance can be operated from a Java-based service component (OSGi bundle). Furthermore, a TuxRacerInputService captures user interaction (e.g. keyboard or multi-touch input) and translates this data into commands to steer Tux (e.g. steer left, accelerate, break, . . . ).

To play Tux Racer, a gamer first registers herself in the environment (e.g. using the ReWiRe client on projector PC) and starts a TuxRacerTask. The PDA then presents her with a user interface (TuxRacerSwingUI) in which an output device – the device that visualizes the gameplay – and a preferred course can be selected as shown in figure 5.11(a). Possible output devices are retrieved from the environment using the SPARQL query in listing 5.7.

```
PREFIX e: <http://edm.org/environment#>
PREFIX tux: <http://edm.org/environment/tux#>
SELECT ?d
WHERE { ?d a e:Device . ?s e:runsOn ?d . ?s a tux:TuxRacerService }

Listing 5.7: SPARQL query asking for devices that run a TuxRacerService.
```

Notice how the upper environment ontology (prefix ‘e:’) and the Tux Racer application ontology (prefix ‘tux:’) are seamlessly accessed in the query. In our test environment, the projector and tabletop PC will be returned as pos-
Figure 5.10: Instance of a Tux Racer application ontology which depicts the setup of the environment.

sible output devices, but other devices can be integrated as well without the need to reprogram services or restart ReWiRe clients. Once an output device is selected, a list of available courses is retrieved from the TuxRacerService running on the output device. After pressing ‘play’, a game instance initiates on the selected output device and the user can start playing. A default TuxRacerInputService on the output device pipes user input to the game instance, but remote input services can be used as well. For example, by executing the TuxRacerInputTask on a PDA, the user is presented with an interface allowing her to drag an iconic representation of Tux back and forth over the PDA screen, as illustrated in figure 5.11(c). Hence the PDA is used as input device while the projector PC renders the game visuals.

Users can also compete with each other in a race. A user can join a game instance by selecting the same output device as another user. In this case the game adjusts to split screen mode which is visualized according to the device the game is rendered on. For example, on a tabletop display, the
5.7 Case studies

(a) Configuring a game instance. (b) Competing each other in mirrored multiplayer mode.

(c) Steering Tux on a PDA. (d) Tux Racer on a multi-touch table.

Figure 5.11: Playing Tux Racer using heterogeneous devices.

split screen view is mirrored (see figure 5.11(b) and (d)) and thus orientend towards gamers sitting on both sides of the table. The name of each player is also displayed during a game. This illustrates the continuous link between model and architecture which is specific to any pervasive application created using ReWiRe: context information (e.g. the user’s name and device type) is propagated to a TUXRACERSERVICE so that it can properly configure a game instance and changes in the environment configuration that have an impact
on the game (e.g. a player suddenly leaving the environment) are reflected in
the model and communicated back to a TuxRacerService so that it can react
appropriately, i.e. terminate the player’s game instance or switch to single
player mode again.

5.7.2 SPOT applications

We developed a number of SPOT applications, shown in figure 5.12 that
connect pervasive services with SunSPOT on the fly. A SunSPOT (see fig-
ure 5.13(a)) is an embedded computing device with a number of built-in sen-
sors such as temperature and light sensors and 3D accelerometers. SunSPOTs
communicate over a wireless network with a base station that is connected over
USB with a desktop computer. We attached the base station to a notebook
on which we installed the ReWiRe client platform. A Java MIDlet running
on a SunSPOT collects sensor readings on this device and sends them to a
SunSPOT service encapsulated in an OSGi component on the client platform
(see figure 5.13(b)). The SunSPOT service then transforms the received infor-
mation into ontology-compliant sensor events such as \{SunSPOT1, TiltSen-

tor\}. From this point, we can leverage the environment and behaviour models
to connect the events produced by a SunSPOT with actions that control a per-
vasive application. We designed a prototype application for automating the
lights in an environment and used a SunSPOT as input device to operate a
media player and play the Tux Racer game. These applications import on-
tologies describing their application domain, integrate the necessary resources
in the environment and subscribe to SunSPOT-specific sensors.

SPOT Lights

The SPOT Lights application integrates a light service that embeds application
logic to turn on/off lights in the pervasive computing environment, a number
of virtual light resources and a user interface for operating lights. In the
application’s user interface, a light can be selected and turned on/off. The user
interface logs user actions that alter the state of a light and links them with
events that might be triggered. Furthermore, a small piece of JavaScript code
injects two ECAA rules in the behaviour model that connect a SunSPOT
with a light resource. These rules automatically turn on/off the light if the light
sensor readings of the SunSPOT drop below or rise above a predefined value.
The script also defines the conditions and actions that are executed by the rule

9 http://www.sunspotworld.com/
5.7 Case studies

(a) SPOT Lights.  

(b) SPOT Media.  

(c) SPOT Racer.

Figure 5.12: SPOT applications are regular applications that can be operated using a SunSPOT.

...
Even though SPOT Lights is not a complex application, it illustrates that runtime debug tools are useful instruments to deal with the complexities introduced by the pervasive computing environment it operates in.

### SPOT Media

The SPOT Media application consists of a media service, a user interface to operate this service and a behaviour script that maps SunSPOT input events on media controls. Tilting a SunSPOT over its X-axis navigates back and forth through the media playlist. A SunSPOT gesture to start playback, e.g. flipping the SunSPOT upside down, can be configured in a dedicated user interface that generates behaviour rules in the background.

### SPOT Racer

In the SPOT Racer application we connected SunSPOTs with the Tux Racer service introduced in section 5.7.1. We created an additional service that passes SunSPOT input events it receives over the network to a game process. Tilting the SunSPOT steers and (de)accelerates Tux while he slides down a ramp full of snow and herring. The tilt events generated by the SunSPOT are connected to behaviour rules that map tilt information, via a script, to navigation commands defined in a Tux Racer service. Similar to the SPOT Lights application, the execution of the game can be monitored at runtime. Although it is not the most efficient approach to encode real-time interaction
5.8 Discussion

In this chapter we have presented ReWiRe as a middleware for developing and deploying pervasive applications. ReWiRe builds on decentralized models that were introduced in chapters 2 and 3 to capture the context of use and describe the behaviour of applications. In ReWiRe, the configuration of the environment is linked with the software architecture: when the environment configuration evolves, software components are ‘rewired’ accordingly. This is achieved by associating context changes with rewiring operations such as the

Figure 5.14: A script adds rules to the behaviour model which connect a SunSPOT with a light resource.

using behaviour scripts/rules, it can be used to prototype the game. As expected, some lag was noticeable on a dual core 1.8 GHz notebook with 2 GB of memory, but it did not prevent us from playing Tux Racer smoothly using a SunSPOT.

```javascript
importBundle('Bundles.org.edn.service.light.proxy');
importBundle('Bundles.org.edn.service.sunspot');
var light;

this.onload = function() {
  onSetup();
  if (light == null) {
    proc.out.println("No light selected, exiting...");
    proc.exit();
  } else {
    var ev = new BEvent("", LightSensor.URL, "SunSPOT light sensor");
    var condDark = new BCondition("isDark", "Is darkness sensed?");
    var condLight = new BCondition("isLight", "Is lightness sensed?");
    var action = new BAction("turnOn", "Turn light on");
    var actionOff = new BAction("turnOff", "Turn light off");
    proc.addRule(new BRule(ev, condDark, action, actionOff));
    proc.addRule(new BRule(ev, condLight, actionOff, action));
  }
}

this.isDark = function(ctx) {
  proc.out.print("isDark?");
  var s = new LightSensor(); s.fromContext(ctx, getParameters());
  var b = (s.getValue() <= dt);
  proc.out.println(" + b + ", s.getValue());
  return b;
}
```
distribution of a user interface to another device. ReWiRe does not dictate requirements for service and user interface components but rather describes the technology that was used to create them in separate groundings. This provides devices with the flexibility to look up a suitable software component that is compatible with its hard- en software platform and download it on the fly. In particular, we used a combination of ontologies (model) and OSGi bundles (software components) to support the deployment of services and user interfaces.

Using ReWiRe we have successfully designed prototype applications that can be observed at runtime by means of the messages they exchange, the properties they manipulate and the rules they define. In particular, we have shown that existing applications can be augmented with pervasive software components and operated using heterogeneous devices they were not designed for. Moreover, we have illustrated that the pervasive computing environment can recover from mistakes caused by system-driven behaviour. For example, if a light was turned off without the user wanting this to happen, the user can still intervene and manually correct an unwanted state.
Chapter 6

PerCraft: Crafting Observable Applications

Contents

6.1 Introduction ........................................ 85
6.2 PerCraft design methodology and tools .......... 86
   6.2.1 Step 1: Modelling ............................. 87
   6.2.2 Step 2: Development .......................... 89
   6.2.3 Step 3: Deployment ......................... 89
   6.2.4 Step 4: Testing ............................... 92
6.3 Case study: Globetrotter .......................... 93
6.4 Discussion ........................................ 95

6.1 Introduction

Hiding the heterogeneity and distributed nature of a pervasive computing environment, e.g. by abstracting the network layer or the deployment process of an application’s services [Grimm 04], introduces caveats at different stages in a pervasive application’s development process and life cycle which are often neglected:

- Performance: hiding essential aspects of a pervasive computing system should be avoided when it constrains developers in taking care of the efficiency of their applications. For example, uncontrolled event subscriptions could disrupt an efficient regulation of event streams and actions that are performed upon receiving an event [Kumar 05].
• *Deployment*: when services are installed beforehand they give rise to pervasive applications after discovery when brought together [Zhu 05]. However, installing a new pervasive application while the system is in use demands for live deployment of services over available devices. Kawsar et al. [Kawsar 08] have pointed out that users benefit from a do-it-yourself approach to deploy applications in their smart homes.

• *Debugging and verification*: if an application does not behave as expected, it is often difficult to attribute a cause to this behaviour without closely inspecting services individually. Moreover, it can be hard to verify whether a pervasive application behaves as expected in the first place. Tools such as PerViz [Pham 10] that ease the debugging process of pervasive applications become increasingly important. Developers interact with PerViz through a Web browser which provides a centralized location to study and filter aggregated application logs and state.

With PerCraft we provide a design strategy for pervasive application which addresses these topics. Instead of hiding a pervasive application by transparently embedding it in the environment, we aim to deliver observable applications whose state can be inspected at runtime. PerCraft contributes to an increased visibility of pervasive applications in order to gain better control over migration, performance and error handling. To accomplish this, PerCraft provides a tool that integrates with the ReWiRe framework similar to the tools presented in section 5.6. ReWiRe and the PerCraft tool are used for orchestrating the deployment of an application’s services and monitoring its execution. Note however that other frameworks featuring a context model and service deployment facilities could be used as well in combination with PerCraft.

6.2 PerCraft design methodology and tools

To illustrate how PerCraft addresses the design of pervasive applications, we will outline the different steps involved in our approach using a prototype application called Globetrotter. The application provides a messenger-like user interface listing friends along with information about their current location. To protect the privacy of its users, the granularity of spatial detail that is published about a user can be configured at a friend-per-friend base. For each friend, the user assigns a profile that determines the level of spatial detail that friend has access to: None (default), Country, City or Spot. The Country and City profiles respectively return the name of the country or the city the user
6.2 PerCraft design methodology and tools

Figure 6.1: Globetrotter integrates real-time location data in a messenger-like pervasive application and is created using PerCraft. The application can be installed by end-users through a setup wizard, similar to a desktop application. Its services are deployed over several devices and might need to be rewired at runtime to maintain an acceptable performance.

currently resides in, whilst the Spot option grants a detailed read-out of the user’s whereabouts (e.g. provided as GPS coordinates). Two plugins extend the Globetrotter application:

1. The TripIt plugin adds support for travel itineraries planned by a user. When a foreign travel destination is reached, it provides access to practical local information such as important phone numbers and cultural habits.

2. The FriendSpotter plugin notifies a user if one of her friends is in the vicinity, e.g. in the same city or within a predefined distance.

The next sections elaborate on the modelling, development, deployment and testing of this application using PerCraft and ReWiRe.

6.2.1 Step 1: Modelling

In the modeling step, we break down an application in a set of services for which we identify sensors and aggregators. As outlined in section 2.4, sensors and aggregators respectively produce and collect context information which might give rise to a new application state. Figure 6.2 shows an application
model for the Globetrotter application. The application is realized using four services of which the TrotterService is the most complex one. As shown in figure 6.2(a), this service defines a number of sensors and aggregators that produce and collect information about a user’s location. Basically, the TrotterService will filter data it receives from a GPSSensor provided by a LOCATIONSERVICE and e.g. transform GPS coordinates into the name of a country using the GMAPSSERVICE and then trigger a COUNTRYSENSOR. Figure 6.2(b) shows the COUNTRYSENSOR in more detail. Just like the CitySensor and SPOTSENSOR, the COUNTRYSENSOR is linked with the User concept (which is also a RESOURCE) and inherited from a more generic LOCATIONSENSOR. Hence one can simply subscribe to a \{UserMe, LOCATIONSENSOR\} event to get notified about changes in the user’s location. Depending on the granularity of spatial detail the subscriber has access to, the LOCATIONSENSOR will be substituted by e.g. a COUNTRYSENSOR when a location event takes place. Globetrotter’s TripIt plugin is supported through the TRIPITSERVICE whilst we provide the FriendSpotter feature using an additional sensor in the TrotterService, namely the FRIENDNEARBYSENSOR.

![Diagram of services, sensors, and aggregators](image)

(a) Services, sensors and aggregators.

![Diagram of relation between resources and sensors](image)

(b) Relation between resources and sensors.

Figure 6.2: Application model for Globetrotter.

The deliverable of this step is an application ontology describing an application’s services, sensors and aggregators. Like any domain ontology, it can be imported in ReWiRe and populate an environment’s context model.
6.2.2 Step 2: Development

In the development step, we implement the services and in particular their sensors and aggregators that were identified in the previous step. Sensors and aggregators are integrated in a service by implementing a Sensor and Aggregator interface respectively. Aggregators use queries in their back-end such as the query in listing 6.1 (MyCity aggregator).

\[
Q_r : \text{SELECT } ?l \text{ WHERE } \{ ?l : \text{location} : \text{TrotterService1} \}
\]

\[
Q : \text{SELECT } ?c \text{ WHERE } \{ : \text{User1} : \text{city} \ ?c \}
\]

Listing 6.1: To select the city a user currently is in, the user’s location context is first retrieved.

The query \(Q_r\) first aggregates the location context of a user in a temporary model and next, the query \(Q\) selects the user’s current city from this temporary model. Note that the city property is only available if the query requestor (i.e. the executor of the aggregator) has sufficient privileges to access this information. Sensors, on the other hand, are passed updated context information and are emitted as events (i.e. resource/sensor pairs with output values such as \{UserMe, CitySensor, ‘Tokyo’\}) over the network to interested parties. Additional application logic is added to realize a service to interface with a GPS chip (GPSSERVICE) or functions to configure a user’s privacy (TrotterService). The GMapsService acts as a wrapper for the Google Maps API. Likewise, the TripItService uses the TripIt API to retrieve trips managed in an online TripIt account.

The outcome of this step is a set of services, encapsulated in deployable packages (e.g. OSGi bundles) that can be deployed on a target device (e.g. using ReWiRe).

6.2.3 Step 3: Deployment

In the deployment step, we first connect model and implementation. Each service is mapped on an OSGi bundle, and each sensor and aggregator on a class in this bundle. This process is semi-automated using reflection, sensors and aggregator classes are resolved and matched with their representations in the model. For example, the TrotterService is linked with the bundle globetrotter.jar and its CitySensor with a Java class inside this bundle:

\[\text{http://code.google.com/apis/maps/}\]

\[\text{http://www.tripit.com/developer/}\]
To manage the deployment of an application’s services, we classify services as zombie, peasant or wizard service:

- **Zombie service**: The service is maintained and served by a third party. Although its deployment is beyond our control, we can write aggregators and sensors for the service, packed in a peasant service.

- **Peasant service**: The service runs on a specific device where it processes either local data (e.g. user input or sensor readings from built-in hardware) or remote data (e.g. to update the state of a local resource).

- **Wizard service**: This service coordinates peasant services and/or includes a user interface (i.e. wizard) to (re)deploy a pervasive application.

Furthermore, we consider two properties – migratable and optional – that can be attributed to any peasant service; a wizard service is nor migratable, nor optional by design:

- **Migratable peasant service**: It does not matter where this peasant service runs to make the application work. However, the device it runs on could have an impact on the application’s performance.

- **Optional peasant service**: The service is part of the application but not really needed to make it work. When available, it extends the application with extra features.

Each application built using PerCraft consists of one wizard service and a number of peasant services. So far, we only created peasant services for the Globetrotter application of which the TrotterService is migratable since all it needs is a CPU and some memory to process incoming sensor events and publish outgoing sensor events. Note also that the GMapsService and TripItService are optional peasant services which rely on zombie services (i.e. Web services) in their back-end. The PerCraft tool depicted in figure 6.3 eases the development of a wizard service. It allows users to drag services on target devices and indicate a master device, i.e. the one that will run the application’s wizard service. When pressing the deploy button, a deployment descriptor is generated from user input and bundled with a wizard service which is sent to the selected master device. A default wizard service interprets the deployment descriptor and migrates peasant services over selected devices, depicted in the descriptor. However, further orchestration is often needed to connect and configure peasant services. This is achieved through a custom wizard service, tailored to the needs of a particular pervasive application. We
6.2 PerCraft design methodology and tools

distinguish between two (re)deployment strategies that can be provided by a wizard service:

- **User-driven**: a deployment wizard (user interface) asks the user about application-specific details. The wizard transforms user input into configuration properties which are sent to appropriate peasant services.

- **Application-driven**: the wizard service automatically ensures that an application runs in optimal conditions even when the configuration of the environment evolves (e.g. devices (dis)appear) or gracefully suspends the application’s services upon failures.

In a custom wizard service for the Globetrotter application, we combined both strategies. A built-in wizard lets a user manually rewire a TrotterService to another suitable device while the application is in use. Furthermore, the application’s wizard service is programmed to gracefully shutdown the application’s services upon failure. For example, if a TrotterService fails (i.e. becomes unresponsive), the wizard will suspend its related GPSService as well. As such, a wizard service basically acts as the brain of an application. While a migratable peasant service can suggest a rewiring action, e.g. because the battery of the mobile device it runs on is almost dead, it is the wizard service that will take final decisions regarding the (re)deployment of an application’s peasant services and their orchestration.

**Figure 6.3**: Generating a deployment descriptor to configure the distribution of peasant and wizard services.

To manage the deployment and life cycle of individual services, PerCraft relies on `install`, `uninstall`, `update`, `start` and `stop` operations with similar semantics as those defined in the OSGi specification [Alliance 09]. These
operations are implemented in a PerCraft client component (e.g. a ReWiRe client) installed on devices and can be remotely invoked by wizard services and end-users to distribute a pervasive application over devices and orchestrate service compositions. Moreover, dynamic service compositions give rise to collaborative applications: when wizard services can be discovered, applications can synchronize their goals and collaborate. In fact, interaction between wizard services might be essential in an application-rich pervasive environment in order to avoid that applications start competing for resources. Rather than optimizing the deployment of services on an application per application base, a group of wizard services could agree on the best compromise to deliver an acceptable performance for each application in the environment.

The deployment step results in a refinement of the previous two steps. Model and implementation of a pervasive application are tweaked to make the application deployable.

6.2.4 Step 4: Testing

In the testing step, we focus on monitoring and benchmarking the flow of context information within an application. While an application executes, the PerCraft monitoring tool visualizes context produced and collected by sensors and aggregators in its peasant services on a real-time chart, shown in figure 6.4. Each sensor and aggregator is attributed an estimated weight that indicates the cost for processing it. Factors that influence this cost are for example its network load and required processing power. Note that when selecting a sensor or aggregator on the chart, its outputs and the service and device from where it originates are displayed. As context updates have a major impact on an application’s behaviour, they help to explain and thus debug a pervasive application. For example, we can trace back a FRIENDNEARBYSSENSOR to a LOCATIONSENSOR which was triggered as a reaction to a GPSSENSOR, analyze their outputs and inspect from which services and devices they originate. Moreover, we can write unit tests as a special type of aggregator (e.g. with a boolean output) to assert that an application behaves as expected at strategic places in its life cycle. For instance, a unit test aggregator that verifies the correctness of a FRIENDNEARBYSSENSOR can reuse and execute the MYCITY or MYSPOT aggregator on your and your friend’s TROTTERSERVICE and assert that the collected location data indeed matches the ‘nearby’ definition. The context flow in an application is also a measure for its performance. For example, deploying a TROTTERSERVICE at the same (mobile) device as a LOCATIONSERVICE is the most efficient when no real-time location coordinates
6.3 Case study: Globetrotter

Figure 6.4: Analyzing the flow of context information in a pervasive application. The estimated cost of processing sensors and aggregators in a service is shown in a chart; output parameters can be further inspected at execution time.

are to be sent out (i.e. all friends are assigned a City privacy level or less). However, when an online friend has access to Spot level spatial data, it could be more efficient to migrate the TrotterService to a more performant device; location coordinates need to be sent over the network anyway. Using PerCraft, we can evaluate at runtime how devices and the services they run can be optimally orchestrated to deliver a desired quality of service such as a minimal network load. Moreover, wizard services can use quality of service measurements to automatically re(deploy) migratable peasant services at runtime and optimize an application’s performance.

By analyzing the context flow in the Globetrotter application, it can be seen that the FriendNearbySensor leads to inefficient behaviour when two users are far away, i.e. if they have Spot level access to each other’s location. In this particular case, the location details of a user residing in Tokyo are continuously exchanged with the TrotterService of a user in New York and their relative distance is calculated to check whether they are not yet ‘nearby’. A more efficient approach would be to poll for location updates instead of listening to LocationSensor events as soon as two users are separated more than a certain distance from each other.

6.3 Case study: Globetrotter

To evaluate PerCraft, a prototype of the Globetrotter application was created as outlined in section 6.2. Since mobile devices are heterogeneous in terms of the software they support, we will address the deployment and management of both tailored components that are closely integrated with a specific device and
loosely coupled components for multiple platforms. For this purpose, we support two PerCraft implementations that can be used to deploy Globetrotter’s services:

- A ReWiRe client for PerCraft is built using OSGi technology and runs on devices that support Java. Due to the close match of PerCraft’s main operations and OSGi’s bundle operations, ReWiRe readily provides the required PerCraft operations. In addition, it supports operations for reporting about the context flow in services published on a device and as such enables the use of PerCraft’s monitoring tool.

- An Android client prototype for PerCraft is specifically targeted at Android devices. Although the Android platform does support OSGi, we opted for a minimal Android client that implements the PerCraft API. Services are delivered in custom packages that can be loaded using reflection by an Android client.

With various types of target devices and dedicated clients in place, pervasive applications designed using PerCraft can provide service implementations for different platforms of which a suitable one is selected when a (re)deployment action is triggered by a wizard service. It is left to a PerCraft client (e.g. a ReWiRe client or Android client) to resolve a suitable service implementation, e.g. using a grounding mechanism as discussed in section 2.3.5.

Next to services, user interfaces are required to interact with an application. Several pervasive frameworks treat user interfaces as a special type of services, in which case they can be considered as (optional) migratable peasant services in PerCraft. Opposed to this, infrastructures such as ICrafter [Ponnekanti 01] and the Personal Universal Controller (PUC) [Nichols 06] handle user interfaces as separate components that can be (semi-)automatically generated. In our prototype implementation, we opted for a separate Web interface served by a small centralized Web application that manages user accounts and connects with a.o. TROTTERSERVICE instances in its back-end. Deploying a user interface for Globetrotter on an end-user device initiates a Web browser at the target device which is pointed to the URL of Globetrotter’s user interface. We opted for a Web interface to support many platforms at once, but richer platform-specific user interfaces that better exploit a device’s input capabilities can be embedded in a PerCraft application as well and deployed similar to services.

3EZDroid (http://www.ezdroid.com/) is an OSGi implementation running on Android
6.4 Discussion

Using the PerCraft tool depicted in figure 6.3, we deployed an instance of Globetrotter over an Android phone and a desktop PC. Other instances of the application were deployed on some of our colleagues workstations as to populate the environment with a number of users. To simulate movement of these users, we slightly modified their application instance with a custom location service that produces random location events.

6.4 Discussion

In this chapter we have presented a design methodology for pervasive applications. With PerCraft we suggest pervasive applications become end-user products which can be (re)deployed over several available devices, either automatically or assisted by the end-user. Similar to desktop applications, a context-aware wizard bundled with a PerCraft application guides the user through the setup process which is re-invoked whenever the configuration of an environment changes in such a way that the execution of an application nor its performance is threatened. By modelling the exchange of context information as aggregators (on demand) and sensors (on change), we have illustrated using a prototype application that tools can be used to inspect the flow of context information within an application at runtime and hence get insight in the best distribution pattern of an application’s services. The major merit of our approach is a simplified deployment of pervasive applications as a whole (i.e. services are not installed individually) whilst still allowing for dynamic service compositions. We acknowledge that developers still need to provide different versions of a service when multiple target platforms are considered. We believe that this situation can only be resolved when a standardized service platform becomes available that runs on recent set-top boxes, Android devices, Windows Mobile phones, etc. Live deployment of services over end-user devices also brings up privacy and security issues which still need to be addressed.
Chapter 7

Semantic Web Applications

Contents

7.1 Introduction .................................................. 97
7.2 OWL notation in JSON ....................................... 99
7.3 SemSon Web toolkit ......................................... 101
  7.3.1 Importing and querying ontologies ................. 101
  7.3.2 Creating and validating individual objects ......... 102
7.4 Case study: semantic user profiles ...................... 104
7.5 Discussion .................................................. 105

7.1 Introduction

The shift to dynamic Web applications running in a Web browser – also known as AJAX applications – has brought up the need to interact with remote data from client-side scripts. At the same time, an increasing amount of meaningful information is published on the Web, making it possible for Web applications to better understand and satisfy requests between users and machines as envisioned by the semantic Web [Berners-Lee 01]. Due to the emerge of Web browsers to end-user devices, responsive Web interfaces are also suitable candidates for interacting with pervasive applications. Since most Web-based applications still rely on a fixed database schema, the semantics of the application data are often known in advance and interaction with a Web server’s database can be abstracted using data objects. However, when developers start to adopt ontologies as a means to structure, query and share informa-
tion, the full semantics of available data are not always known beforehand. One ontology can extend another ontology with extra concepts and relations and hence give rise to a richer knowledge base. For example, in the semantic environment model presented in chapter 2, domain ontologies are aggregated with an upper ontology on the fly. In this case, data objects that are used to retrieve and update available information should adapt to match the concepts in the aggregated knowledge base. In ReWiRe for example (see chapter 5), Java classes for interacting with new domain concepts are dynamically added by loading corresponding OSGi bundles. Unlike Java, however, JavaScript – the default programming language for client-side Web applications – is a scripting language and differs in the way objects are composed.

With the SemSon framework, we aim to bridge the gap between semantic data defined in OWL ontologies and client-side Web applications implemented in JavaScript. SemSon focuses on the dynamic use of OWL constructs in JavaScript to make meaningful information directly available to client-side Web applications. The main difficulty to link these technologies lays in the fact that ontology experts think in terms of concepts and relations while many software developers are used to object-oriented programming languages (OOPLs). Koide et al. [Koide 06] explain the semantic gap between OOPLs and OWL: various discrepancies between static OOPLs such as Java and OWL/RDF are due to a mismatch between OWL classes/individuals and OOPL classes/instances:

- Each instance in OOPLs belongs to one class; unlike in OWL, multiple class inheritance is often not supported in OOPLs.
- The list of classes in OOPLs must be fully known at compile-time and cannot change after that.
- There is no reasoner in OOPLs that can be used for classification and consistency checking at runtime or build-time.

The authors of [Koide 06] show that these mismatches can be addressed by using a more dynamic and reflective OOPL such as Common Lisp Object System (CLOS) and present an OWL processor built on top of CLOS that allows programmers to develop application domain models using OOP. A more static approach for mapping an OWL ontology on Java classes and instances is presented in [Kalyanpur 04]. In this work a solution is proposed to create a set of Java interfaces and classes from an OWL ontology whilst minimizing the impact of fundamental semantic differences. An instance of a Java class re-
7.2 OWL notation in JSON

represents an instance of a single class of the ontology with most of its properties, class relationships and restriction definitions maintained.

With SemSon we focus on the dynamic use of OWL constructs in JavaScript to make meaningful information directly available to client-side Web applications. JavaScript, the default programming language to develop this type of applications, is just like CLOS very flexible in the way objects are composed:

- Multiple class inheritance: properties (and methods) can be inherited from multiple classes.
- Dynamic programming: JavaScript allows to redefine and extend classes with new properties and methods at runtime.

A growing interest of JSON in combination with semantic Web technologies such as SPARQL also motivates our research. For instance, SPARQL query results can be serialized into JSON and thus processed efficiently in JavaScript through object-oriented programming [Clark 07].

7.2 OWL notation in JSON

As a first step in enabling the use of semantics inside a Web browser we define a mapping of OWL syntax on JavaScript syntax and vice versa. We connect both worlds by representing information defined in an OWL ontology using class objects and individual objects, expressed in JSON. Class and individual objects correspond to OWL class and individual descriptions respectively as depicted in figure 7.1. In both representations, URIs (namespaces and identifiers) are used to refer to objects (JSON) and resources (OWL). Note however that SemSon is not an API for OWL, but rather provides a means to generate JavaScript objects from OWL DL semantics. We rely on the DL subset of the complete OWL vocabulary because it enforces a strict separation of classes, properties, individuals and data values which is also found in OOPLs.

Figure 7.1: Mapping OWL to JSON.
In the remainder of this section we give a brief overview of the representation of OWL class descriptions in JSON and section 7.3 elaborates on the creation, use and validation of individual objects.

OWL classes can be described using a simple class identifier (URI reference) or built using the following constructs for which we provide a JSON equivalent:

- **Enumeration**: an OWL class can be described by exhaustively enumerating its individuals.

  ```json
  {"owl": {"oneOf": ["A", "B"]}}
  ```

- **Intersection, union, complement**: AND, OR and NOT operators can be applied to OWL class descriptions.

  ```json
  {"owl": {"intersectionOf": ["A", "B"]}}
  {"owl": {"complementOf": "A"}}
  ```

- **Properties**: properties are linked with class descriptions through domain and range axioms. Restrictions on properties put additional constraints on the range of an OWL property when applied to a particular class description (value constraints) or on the number of values a property can take (cardinality constraints).

  ```json
  {"owl": {"properties": [    {"uri": "P", "range": ["A", "B"], "cardinality": "1"},    {"uri": "Q", "hasValue": ["C"]}]}}
  ```

OWL contains three language constructs for combining class descriptions into class axioms. These axioms describe inheritance, equivalence and disjointness relations that exist between classes.

```json
{"owl": {"subClassOf": ["A"],     {"equivalentClass": ["B"],     {"disjointWith": ["C"]}}
 ```
This information enables a developer to reason about the class hierarchy and semantically compare classes.

## 7.3 SemSon Web toolkit

The SemSon framework consists of a JavaScript library (semson.js) and a collection of Java-based servlets (semson.war) which connect scripts running in a Web browser with ontologies published on the Web, as shown in figure 7.2. We use Jena\(^1\) to create a semantic context store, Pellet [Sirin 07] as OWL reasoner and SPARQL [SPARQL 08] as query language. While we made a clear distinction between the semantics of resources and their instance data in the distributed context store discussed in section 2.4, we now assume that ontologies and their instances co-exist in a shared repository. In this case, a Web application can directly use the context store as a semantic database to store both ontologies and their instances, which reduces the complexity of SemSon’s architecture. A reasoner such as Pellet ensures that instances that are added or updated to the context store are consistent with their class definition(s). Table 7.1 lists the main operations supported by SemSon.

### 7.3.1 Importing and querying ontologies

Data is made available to Web applications through a semantic context store which acts as an application’s database. Instead of using a fixed database schema, SemSon allows programmers to specify the ontologies they want to use at runtime and dynamically loads them into the context store. Once in the context store, information from the ontology can be derived by constructing JavaScript objects that correspond to a class or individual whose related data

---

\(^1\) http://jena.sourceforge.net/
Table 7.1: Main SemSon operations.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>import(location)</td>
<td>Import an ontology into a context store.</td>
</tr>
<tr>
<td>c(c_uri)</td>
<td>Create a class object from an OWL class.</td>
</tr>
<tr>
<td>i(c_uri,i_uri)</td>
<td>Generate an individual object from an OWL class.</td>
</tr>
<tr>
<td>validate(i,c)</td>
<td>Validate an individual object against a class object.</td>
</tr>
<tr>
<td>update()</td>
<td>Receive individual data from a context store.</td>
</tr>
<tr>
<td>commit()</td>
<td>Send individual data to a context store.</td>
</tr>
</tbody>
</table>

is transparently retrieved from the context store or by using SPARQL queries when more specific data selection is needed. A preprocessing step generates native JavaScript object wrappers for each OWL class description in an ontology. These wrappers support the creation of objects in an object-oriented way: `new MyClass()` is equivalent with `semson.i('myclass')`. When preprocessing is omitted, e.g. when dealing with large ontologies with many class descriptions, class objects are resolved on an as-needed basis.

### 7.3.2 Creating and validating individual objects

In SemSon, objects are automatically generated from an OWL class description at runtime. Software developers only need a minimal understanding of the ontologies designed by domain experts such as available classes and their properties. Objects abstract underlying (instances of) ontologies and thus simplify working with semantic data. Furthermore, when an ontology is modified, data objects are updated accordingly: the application developer gets an updated data interface for free. Consider for instance an object that represents a person described in an ontology with namespace `ns`. This object is created as follows:

```javascript
var p = new ns.Person(); // eq. semson.i('ns:Person');
```

To construct an instance of a class, we analyze the class description fetched from the ontology as described in section 7.2 and dynamically generate an object whose member variables match the OWL properties and their restrictions. For example, if a restriction on the ` ns:PERSON` class defines that a person has zero or more hobbies, we generate a `hobbies` member variable that corresponds to an array:
Moreover, we have to take into account class descriptions based on intersection, union and complement as well as axioms such as rdfs:subClassOf when generating objects. These constructs specify the individuals that are supported by a class and thus also indicate valid properties for instances of classes. We select a superset of properties which allow to create valid class instances and map these on object member variables. Whether an instance is compliant to a class or not can be validated at runtime (see further). To avoid clashes with member variables, we use namespaces to differentiate between similar properties, e.g. obj.property versus obj.ns.property.

Most restrictions on properties and classes are hard to enforce while generating an individual object. An individual object can be manipulated at runtime and become incompatible with its class object. To detect inconsistencies, SemSon includes a validation algorithm implemented in JavaScript – validate(i,c), with i an individual object and c a class object – which compares the data of an individual object with its corresponding class object. The algorithm currently consists of three steps:

1. **Enumeration checking**: check whether i equals one of the allowed individuals for c.

2. **Property checking**: verify whether properties declared in an object are allowed, have valid values and a correct number of elements. This also involves class and datatype checking, for instance to assert that a property value indeed matches a class or value within a specified range.

3. **Intersection, union, complement checking**: assert that an object passes the following tests: validate(this,A) && validate(this,B), validate(this,A) || validate(this,B), !validate(this,A)

The validation algorithm runs in a Web browser and is fast, but it does not carry out advanced reasoning techniques or complex datatype checking (e.g. user defined schema types). Its main purpose is to provide an efficient means to debug client-side Web applications by asserting that data objects are valid at strategic places in the program. Besides, the algorithm can be used to quickly validate user input by encapsulating input data in an individual object and performing validation. However, to make sure a semantic context store remains consistent, a more powerful approach towards consistency checking is needed. This is achieved by installing an OWL reasoner such as Pellet.
between query engine and context store which analyzes data before making it persistent.

A class or individual object retrieves data from the context store by invoking its update function and data is sent back to the context store by executing the commit function. For performance reasons, we implemented these methods in SemSon servlets deployed on a Web server which have direct access to the context store. An update extracts data from the context store using the resource’s URI as a reference and maps this information on JSON as depicted in figure[7.1] At the client-side script, the JSON data is converted into an object and convenience methods are added. A commit sends an object serialized into JSON to the Web server where it is checked by a reasoner for consistency and finally added to the context store. We also support an atomic commit operation in SemSon using transactions. In a transaction, multiple individual objects can be grouped together and committed to the context store in a single operation.

7.4 Case study: semantic user profiles

As a proof of concept, we have developed a prototype Web application in which we use an OWL DL version of the Friend of a Friend (FOAF) vocabulary to create user profiles, as shown in listing[7.1] The FOAF ontology is imported into the context store and is preprocessed so that OWL classes and individuals can be directly instantiated using OOP. In this example, we use a transaction to send the new and updated user profile to the context store and verify the information was stored correctly using a SPARQL query. When the ontology is extended with constraints on properties which e.g. state that each person must have a valid name, we can validate individual objects at runtime. Furthermore, the Pellet reasoner can be used to automatically infer missing information when adding data to a context store. For example, if the FOAF vocabulary is extended with family relations which typically include inverse relations such as A parentOf B ⇔ B childOf A, the reasoner can automatically infer missing facts and add them to the context store to keep ontologies and their instances consistent at all times.

```javascript
var foafns = 'http://www.mindswap.org/2003/owl/foaf';
var friend = 'http://myns/myfriend';
semson.registerNS('foaf', foafns);
semson.import(foafns, true); // register class objects
```
7.5 Discussion

In this chapter we have presented a framework called SemSon to support the development of semantic Web applications that can make use of OWL/RDF data from within a Web browser. Due to the increasingly important role of the (semantic) Web in ubiquitous environments, Web applications running in a browser can also be considered valuable building blocks, in particular if they are targeted toward Web programmers who are familiar with JavaScript, but only have a basic understanding of OWL and ontologies in general. When SemSon is used in combination with dynamic context models such as described in chapter 2 and 3, context-aware Web applications can be constructed. Besides a case study about user profiles, we used SemSon to quickly prototype Web interfaces for use with ReWiRe, discussed in chapter 5. By leveraging ReWiRe’s context store, SemSon-enabled Web interfaces can query a pervasive environment and present the state of its resources, which is particularly useful for debugging purposes. To make such Web interface functional, communication middleware such as a W2P (see chapter 4) can be used to connect the Web interface with distributed services.

Listing 7.1: Creating a FOAF user profile using SemSon.

```javascript
var p1 = new foaf.Person(); // create individuals
p1.firstName = 'first'; p1.surname = 'last';
var p2 = new foaf.Person(friend);
p1.knows.push(p2.uri);
p2.knows.push(p1.uri);
semson.commit(p1, p2); // transaction
var q = 'SELECT ?p WHERE {?p foaf:knows <'+ friend +'>}';
var qr = semson.select(q); // querying
for (b in qr.results.bindings)
    if (qr.results.bindings[b].p.value == p1.uri)
        alert('found!');
```
Part III

Interacting with Pervasive Applications
Chapter 8

Meta-User Interfaces

Contents

8.1 Introduction .................................................. 109
8.2 Universal remotes ............................................. 111
8.3 A view on the environment ................................. 113
  8.3.1 End-user’s view ............................................. 114
  8.3.2 Developer’s view .......................................... 114
8.4 Reference framework .......................................... 115
  8.4.1 Environment model ........................................ 115
  8.4.2 Functional requirements .................................. 115
8.5 Runtime use ..................................................... 117
  8.5.1 Integrating and configuring resources ................ 117
  8.5.2 Proactive behaviour ....................................... 118
8.6 Case study: ReWiRe meta-UI ............................... 119
8.7 Discussion ....................................................... 121

8.1 Introduction

Interaction with computing resources that are invisibly integrated in a physical environment surfaces particular end-user requirements. Users should be able to explore an environment and discover smart resources so that they can interact with them and configure their behaviour, e.g. using a mobile phone:

- Explore: In line with Weiser’s vision on ubiquitous computing [Weiser 91], technology seamlessly integrates with our homes and offices. Intelligent
resources are invisibly embedded in the environment (e.g. as part of the furniture) and aim to assist end-users in their everyday tasks. To interact with the invisible digital dimension of such a computer-augmented environment, users first need to become aware of the technology that is surrounding them, i.e. get insight in the resources that are at hand and the tasks they support.

- **Interact and configure**: Bringing together different resources automatically gives rise to a pervasive environment. For example, combining the user’s mobile phone and the car’s stereo yields a pervasive car environment. This assembly of resources can also enable users to browse the phone’s contact list and initiate calls through the car’s head-up display. However, spontaneous interactions between resources are only an improvement if they can be configured effectively by the end-user. Manual control over the assembly of resources (e.g. the phone and the car’s radio) and their behaviour remains a crucial need to avoid frustrations. People must be able to shape a pervasive environment according to their own demands and preferences [Bellotti 02].

To address these needs, a pervasive computing environment must support a point of control to inspect and manipulate the state of resources. Analogous to desktop systems in which a file explorer and a start menu are indispensable tools, we argue a pervasive computing environment requires similar tools that can be accessed using embedded displays or personal devices. Norman [Norman 02] suggests several design principles that ideally should be met for every device and appliance: visibility, feedback, natural mapping, constraints, and design for error. We aim to improve the visibility of resources in the environment with a pervasive start menu and leave room for end-user tools that accommodate other requirements such as feedback and error recovery. Simplified versions of the developer-oriented tools that are discussed in part II could fulfill this purpose.

In the next sections, we introduce universal remotes and provide an overview of existing meta-user interfaces which can be considered as approaches towards a pervasive start menu. Coutaz [Coutaz 06] defines a meta-user interface (meta-UI) as follows:

> an interactive system with its own user interface that provides end-users with means for controlling interactive spaces.

The functionality Coutaz attributes to a meta-UI – a point of control for an interactive space – reflects the purpose of the pervasive start menu we envision.
8.2 Universal remotes

New appliances that hit the market typically are simple to use in their physical dimension (e.g. few buttons) but become more complex to handle in their digital dimension (e.g. overloaded user interfaces). Many out of the box products ship with their own remotes that provide access to their features assisted by on screen displays. This leads to situations where we end up with a remote for each smart product we buy, as captured by the cartoon in figure 8.1. Moreover, users often need to carry out tasks that involve more than a single appliance such as switching on the television and DVD player at the same time.

Universal remotes such as the Philips Pronto series[^1] overcome the need for multiple remotes by delivering an integrated user interface for multi-room control. However, many commercial universal remotes merely deal with learning and emitting IR/RF codes, rely on proprietary protocols and lack support for context-aware third-party applications. Although universal remotes are gaining support for Universal Plug and Play (UPnP) [UPnP] so that they can discover devices and their services, it remains difficult for end-users to

[^1]: http://www.pronto.philips.com/
differentiate between similar discovered resources based on a textual description, especially in complex environments with many embedded computing resources. This is due to the fact that the UPnP protocol only returns limited information about discovered entities which we believe is not sufficient to compose a pervasive start menu from. Nichols and Zimmermann et al. developed the idea of using mobile computing devices as a personal universal controller [Nichols 06] or universal remote console [Zimmermann 02]. These systems use smartphones and PDAs to dynamically download a user interface from an appliance.

Meta-user interfaces can be part of a universal remote, e.g. to help locate appliances and request a suitable user interface. They are readily integrated in various research applications [Heider 02, Biehl 04, Nichols 06], but are often closely tied to a specific application domain. Figure 8.2 shows two examples of meta-UIs that are applied to control the distribution of user interfaces across multiple displays and devices. These systems try to provide a natural mapping between the meta-UI and the physical environment and e.g. make it easy for the user to select a particular display or target device using a geometric world model. A more generic approach towards meta-UIs can be found in middleware or roomware systems such as ICrafter [Ponnekanti 01]. In ICrafter, different hardware resources (e.g. lights, cameras, projectors, printers, ...) are targeted and a list of application user interfaces can be requested from the pervasive environment. The main goal of the ICrafter framework is to automatically combine different user interfaces provided by various devices into a single user interface. Another approach to represent and organize resources in a pervasive environment are flow-based interfaces like Huddle [Nichols 06] or the framework proposed in [Newman 02]. These systems allow the end-user to specify the data flow between the resources in a pervasive environment. However, their meta-UIs do not provide feedback about the state of services and various resources that inhabit a pervasive environment such as users are not incorporated.

Olsen et al. describe a minimal meta-UI for pervasive environments in [Olsen 01]. They propose two operations (join and capture) and two types of resources (services and devices) for interacting with a pervasive environment. We extend this minimal set of operations and resources into a generic reference framework for meta-UIs that can act as a pervasive start menu and hence allow end-users to discover resources and applications and interact with them.
8.3 A view on the environment

A meta-UI aims to simplify the role of the end-user (interacting with the environment) and the role of the developer (designing and integrating new resources in the environment). In addition, it gives more power to end-users and allows them to manipulate a pervasive system in ways not possible before. Consistent with our definitions introduced in chapter 2, we use the term ‘resource’ to refer to anything that is present in a pervasive environment, ranging from physical resources (e.g. a light bulb, a lever, a computing device) to virtual resources (e.g. a software component, a task description). The specific role of the developer or designer depends on the type of resource: services and user interfaces are often created by different people. We witness the roles of the end-user and the developer are merging when it comes to integrating new resources and configuring the environment. Services or user interfaces can be programmed to automatically adapt to the context of use, but end-users still need to finetune the configuration of a resource. The meta-UI helps to bridge the gap between the two camps.
8.3.1 End-user’s view

From an end-user’s perspective the meta-UI acts as an instrument to inspect and manipulate the state of one’s surroundings. Hence the main responsibility of the meta-UI is to integrate the features supported by the environment and to present them in an intuitive way to the end-users. We consider two major approaches to provide the end-user with an interactive view on the environment’s features:

- **Service-oriented**: In a service-oriented view, the end-user is shown a list of software services that make up the pervasive computing environment, where she can directly interact with.

- **Task-oriented**: In a task-oriented view the pervasive environment is seen as an integrated system where tasks define what the end-user can do within the environment. The end-user is presented with an overview of available tasks while the actual software services that give rise to these tasks are hidden.

The service-oriented view is a more conventional approach supported by different service discovery frameworks such as UPnP, whilst the task-oriented view is inspired by the observation that users think in terms of goals they want to achieve in the environment [Heider 02]. Consider for example a scenario that demands for multimedia features such as playing music in a room. A service-oriented view might integrate a ‘media’ service that exports an integrated user interface to perform all media-related tasks. In contrast, a task-oriented view might present the end-user with ‘play media’ and ‘create playlist’ tasks, which are represented as separate user interfaces to the functions of an underlying ‘media’ service. The advantages and disadvantages of both approaches affect amongst others the amount of time it takes for developers to integrate new applications and the end-user experience of the meta-UI on various heterogeneous devices. We argue a reference framework for a meta-UI should be generic enough to support both views.

8.3.2 Developer’s view

From a developer’s point of view the meta-UI is a software component with a set of common features: it can discover available resources, integrate them in a view and make their software interface accessible to end-users via a user interface. A generic meta-UI provided as part of a middleware framework simplifies the task of developing new resources for a pervasive environment.
since the developer does not have to care anymore how the resource’s functions are made accessible to the end-users. Instead the developer can concentrate on the individual resources such as functional services, task descriptions or user interfaces and rely on the meta-UI to make these resources available to the end-users.

8.4 Reference framework

To cope with the heterogeneity of resources in a pervasive environment and the lack of a specification to uniformly access and integrate these resources, we introduce a reference framework for meta-UIs.

8.4.1 Environment model

To abstract away from the implementation level, we assume a meta-UI leverages a model that describes the context of the pervasive computing environment. This model acts as a data source for the meta-UI (i.e. the view) and can be queried for available resources and their properties. In particular, the model presented in chapter 2 is a suitable candidate to serve as input for a meta-UI as it readily defines major resources that populate a pervasive environment such as users, devices, services and tasks.

8.4.2 Functional requirements

A generic meta-UI should support a canonical set of operations for interacting with the resources present in the environment [Olsen 01, Coutaz 06]. The operations we propose are independent of any specific underlying middleware, yet rely on an environment model as discussed in chapter 2.

First, the meta-UI needs to be aware of changes that occur in the environment’s configuration and reflect these in its view. Therefore it can monitor the environment model and get notified of events that are triggered when resources enter or leave the environment: a new resource is added to the environment ($R+$) or an existing one is removed ($R-$). In particular, a resource can be a user ($U+$, $U-$), a device ($D+$, $D-$), a service ($S+$, $S-$) or a task ($T+$, $T-$). Apart from updating its view, the meta-UI may proactively propose rewiring strategies, e.g. if a new device becomes available that is better suited to execute a task. However, proactive behaviour is considered an additional feature that can be supported using the operators described next, but which is not addressed directly in the proposed reference framework.
Second, a basic set of operators is required to interact with resources from within the meta-UI:

- **Share**(R), **Unshare**(R): Share or unshare a resource. The resource will become (un)available in the pervasive environment (model). This might have an impact on the availability of other resources. For example, a task can only be executed if the services it depends on are in place.

- **Present**(T), **Present**(T,D): Present a task on a device by means of a user interface. A compatible user interface is distributed and rendered on the target device, e.g., a graphical user interface or a speech-based interface. The task is terminated if its user interface is closed. If no target device is specified, the device running the meta-UI that triggered the operator is considered the target device.

- **Suspend**(T), **Resume**(T): Suspend a task and resume it afterwards. The state of the task and/or the user interface presenting it is stored until the task is resumed.

- **Migrate**(T,D): Migrate a task from one device to another. The task is suspended on the source device (**Suspend**(T)) and its context is transferred to the target device where the task is resumed (**Resume**(T)).

- **Invite**(U,T): Invite a user to execute a task, for instance a task that is associated with a collaborative application. An invite is sent to the user’s (default) device which is extracted from the environment model.

These operators arise from the actions users are accustomed to when interacting with a computing device and its local software platform: start an application, interact with it, save a document, etc. In a pervasive environment, an application is not an isolated software component but a dynamic assembly of resources and in particular tasks. If we consider tasks as the building blocks of a pervasive application, end-users must be able to start and manage them (**Present**(T), **Suspend**(T), **Resume**(T)). Besides, in order to exploit the heterogeneous nature of a pervasive computing environment, end-users must be able to traverse tasks to those devices best suited for executing the task (**Migrate**(T,D)) and collaborate with other users (**Invite**(U,T)). As a pervasive environment is open to different users, a privacy issue arises if all resources are shared by default. Hence, end-users need control over the availability of personal resources such as their mobile phones (**Share**(R), **Unshare**(R)). Additional operators can be composed using this basic set such as an **Invite**(T)
8.5 Runtime use

The meta-UI is used to guide the assembly of resources. It provides support for manually configuring (the role of) resources at runtime. When the context in which (inter)actions are performed changes (e.g., a user switches from one device to another), a smart meta-UI could also proactively initiate a rewiring process. In this case, the meta-UI acts like a generic wizard service in the PerCraft design methodology, presented in section 6.2.

8.5.1 Integrating and configuring resources

Typical pervasive environments are constructed in an incremental way from the bottom up: resources join the environment and give rise to new features [Edwards 01]. Therefore, we expect (software) resources that integrate with the meta-UI to be dynamically upgradeable and extensible such that:

- an application or appliance vendor can easily propagate updates after shipping a product;
- a resource can be extended at runtime with functions provided by a third party manufacturer;
- resources can be assembled to accomplish a shared goal.

If a resource becomes available in the environment, it often needs to be configured before it can be used. Also during usage, a resource may need to be reconfigured, for instance due to changes in the environment configuration. An intelligent environment is typically characterized by the fact that it automates many configuration steps and hides them from the user. However, configuring and connecting resources in a meaningful way requires a profound knowledge about the user’s goals. To make sure end-users are always in control, they must be able to steer the environment’s behaviour. In the meta-UI, we support user-driven configuration at two levels:
1. Configuration of the context of a resource.

2. Configuration of the role of a resource in the environment.

The former, configuring the resource itself, can be considered as a user task and thus can be handled by the Present($T$) operator where $T$ is a configuration task presented on a device by a dedicated user interface. The latter, configuring the role of a resource, is more complicated since a resource’s role depends on the tasks the resource is enrolled in. However, as a task is a resource on its own, configuration can be handled in a similar way as in the first case. Consider for example a pervasive paint application that is integrated in the environment as a ‘paint service’. In this case a configuration task will merely deal with the allocation of resources for certain (sub)tasks the application supports, e.g. migrate a colour palette to a PDA and use a whiteboard as a shared paint canvas.

8.5.2 Proactive behaviour

The configuration of a pervasive environment is altered in different ways, e.g. using the meta-UI but also by means of real-world interactions such as toggling a light switch or pushing the thermostat’s ‘temperature up’ button. Software services can become agents that act on behalf of a user [Shneiderman 97]. When an agent is programmed or trained to sense contextual changes that occur as a result of actions performed in either the digital or in the real world, it will automatically execute a certain behavior in return of a context event. When context events are coupled to the operators supported by the meta-UI, the meta-UI itself becomes proactive because it reacts on changes in the environment. For example, when the presence of a PDA device is sensed near a whiteboard that runs a pervasive paint application, the Migrate($T, D$) operator might be invoked to transfer a part of the application’s user interface to the PDA device (e.g. its color palette). In case such behaviour is not what the user expected, e.g. the end-user accidentally approaches the whiteboard with her PDA, the meta-UI can be used to resolve conflicts. For example, if an application’s behaviour is described by a set of rules as suggested in chapter 3, a tool such as the one presented in figure 3.5 can be used to reverse a rule’s effects. Alternatively, when deployment wizards such as discussed in section 6.2 are considered part of the meta-UI, they can be used to (re)configure the distribution of user interface components.

Furthermore the meta-UI can be exploited as a tool to combine different resources and to setup behaviours. Using the environment model, the meta-UI
8.6 Case study: ReWiRe meta-UI

can query the environment for available interaction resources and assign these to a certain service. Consider for example a display attached to device $D_1$ and a keyboard attached to device $D_2$. In order to allocate these resources to a text editor application, the meta-UI running on a device $D_x$ can execute $Present(T_i, D_1)$ and $Present(T_o, D_2)$ with $T_i$ and $T_o$ tasks provided by a text editor service $S$; $T_i$ collects keyboard input and sends it to $S_1$, $T_o$ receives input data from $S$ and visualizes it on a display. When $D_1$ or $D_2$ leaves the environment, the resource assembly is breached and the meta-UI can propose to allocate the lost task to another suitable device. The meta-UI then behaves proactively but still involves the end-user in the decision process instead of handling it autonomously.

Scaling a meta-UI to the context of use goes beyond adapting its user interface to screen resolutions of heterogeneous devices and their interaction modalities, a topic that is already discussed extensively in literature [Paternò 03, Florins 06]. Adapting the meta-UI also encompasses many other factors from which we highlight two important ones based on our experiences:

- **User background**: Since people are interested in different tasks their view on a similar environment can differ. The meta-UI needs to anticipate this observation by personalizing the view according to the envisioned user tasks and goals.

- **Resource explosions**: A crowded environment with many resources in it is hard to inspect. Especially in unfamiliar environments, it will be hard to locate for instance the nearest printer in a room. To avoid an explosion of resources, spatial information could be taken into account to display only those services and tasks in the user’s vicinity.

8.6 Case study: ReWiRe meta-UI

The meta-UI integrated in the ReWiRe framework which is discussed in chapter 5 provides a task-oriented view on the environment and is shown in figure 8.3. It presents an overview of available resources, organized in a menu tree (e.g. ‘media’ menu), along with the tasks these resources support. If a task is selected by the end-user, the meta-UI will try to render a user interface for the task and integrate it with its built-in task manager. In the remainder of this section we will discuss how ReWiRe deals with the functional requirements of a meta-UI introduced in section 8.4.2. The suggested events and operators are supported as follows in ReWiRe:
(a) Overview of media resources and their supported tasks.

Figure 8.3: The meta-UI in the ReWiRe framework.

• $R^+, R^-$: Upon initialization of the ReWiRe tool, the end-user is presented with a dialog to register herself ($U^+$) and her computing device ($D^+$) in a (discovered) pervasive environment. The services installed on the device will also populate the environment’s context model ($S^+$) and export executable tasks ($T^+$). When user and device are signed out of the environment ($U^-, D^-)$, the services running on the device and the tasks supported by these services become unavailable ($S^-, T^-)$.

• $\text{Share}(R), \text{Unshare}(R)$: Resources are shared by ‘publishing’ them on a computing device; a reference to the resource is added to the environment model and the resource becomes available in the meta-UI. If a resource is not shared, it can only be accessed from its local computing device.

• $\text{Present}(T), \text{Present}(T, D)$: The $\text{Present}(T)$ operator translates a task description into a user interface. When a task is assigned to a device, for instance by selecting the task in the meta-UI or dragging it on a device representation, the meta-UI will look-up a suitable user interface in the model to present the task as discussed in section 5.5. Information about the coupling of tasks and their supported presentations and software dependencies is stored in groundings which are included in the domain ontologies as outlined in section 2.3.5. Since a task can be pre-
8.7 Discussion

In this chapter we have presented meta-user interfaces as a means for end-users to interact with the resources in a pervasive environment, e.g. steer the projector in a meeting room directly from a user’s handheld device. The meta-UI features user-driven configuration of resources and their role in the environment. It allows end-users to combine independent resources in meaningful ways, for instance allocate a projector (output) and a mobile phone (input) to a presentation task. Although a meta-UI for a pervasive computing environment is found in different applications, there is still no agreement on its requirements and functions for which we have proposed a reference framework. Unlike user interfaces for specific tasks, services or appliances, the meta-UI allows users to explore their surroundings, i.e. identify the tasks and services supported by the environment and allocate available resources to execute tasks and interact with services.
With a meta-UI implementation for ReWiRe, we have illustrated how our proposed reference framework can be applied in practice. The ReWiRe meta-UI was successfully used to integrate proof of concept applications in a pervasive environment.
Chapter 9

Pervasive Maps

Contents

9.1 Introduction .............................................. 123
9.2 Modelling the environment using photos ............... 124
  9.2.1 Integrating places, sights and resources .............. 126
  9.2.2 Integrating applications ................................ 128
  9.2.3 Tag and search ...................................... 129
9.3 Exploring the environment ................................. 130
  9.3.1 Application-based navigation ......................... 130
  9.3.2 Resource-based navigation .............................. 131
  9.3.3 Location and orientation tracking ...................... 133
  9.3.4 Augmented reality .................................... 134
9.4 Pervasive Maps framework ................................. 135
  9.4.1 PM host ............................................ 136
  9.4.2 PM client .......................................... 136
  9.4.3 PM toolkit ......................................... 137
9.5 Case studies .............................................. 137
  9.5.1 Museum visit ....................................... 137
  9.5.2 Digital scores ....................................... 139
9.6 Discussion ................................................. 141

9.1 Introduction

With Pervasive Maps, we target a meta-UI that adopts the most concrete and intuitive representations one can find of a pervasive environment: pictures.
Studies have for instance shown that pictures assist people in finding their way in previously unexplored environments [Ross 04, Taher 09]. Interaction with a pervasive environment based on video stills and photographs has also been researched before in [Rekimoto 95, Ferscha 03, Suzuki 05]. The NaviCam [Rekimoto 95] and the Digiscope [Ferscha 03] focus on real-time use and hence can not be used outside the target environment. For instance, the Digiscope uses a semi-transparent tablet that provides users with an augmented reality view of invisible information in the environment. Suzuki et al. [Suzuki 05] annotate a ‘u-Photo’ with eyemarks: physical entities that appear on a photo, representing pervasive services (see figure 9.1). By clicking on an eyemark, a user interface for interacting with the service is overlaid on the photo. u-Photos are generated on the fly by taking pictures of an environment in which ARToolkit markers [Kato 99] are embedded. Whereas u-Photo dynamically delivers annotated photos for interaction with the user’s current environment, Pervasive Maps provides a full desktop interface in which photos are integrated in advance. In our solution, photos are primarily used to build up a model of the environment and in second stage as a means for discovering and interacting with remote resources from mobile devices. We consider services to be part of the underlying architecture, and rather focus on resources embedded in places in the environment’s user interface to closely match the real world situation.

The Personal Environment Controller (PECo) [Nazari Shirehjini 04] incorporates a 3D visualization of a physical environment in conjunction with access to personal media. Devices are represented in the virtual environment as interactive 3D objects and provide access to control user interfaces. The PECo system is similar to our approach in the sense that it relies on a digital representation of the environment to interact with the real world. However, designing 3D models is a complex task which is unlikely to be performed by end-users. In Pervasive Maps end-users design their own virtual environment by taking pictures and annotating them. Moreover, our system is different from PECo in the way resources and applications are treated: a resource is ‘opened with’ a suitable application and is not limited to a UPnP device with an embedded user interface. Instead, we allow passive objects with no computing power to become interactive parts of the virtual environment as well.

9.2 Modelling the environment using photos

We use a dedicated domain ontology in combination with the semantic environment model introduced in chapter 2 as internal data structure to represent the environment context. The PM ontology is built up from PLACE, SIGHT,
9.2 Modelling the environment using photos

Figure 9.1: Visual markers embedded in the environment and a camera-enabled device give rise to interactive u-Photos [Suzuki 05].

**RESOURCE** and **APPLICATION** concepts as depicted in figure 9.2.

Figure 9.2: PM domain ontology.

Resources are organized into places that make up the environment and sights provide views on the various places and their embedded resources. Furthermore, a resource can serve as input to an application, similar to opening a file with a suitable application. Since resources are diverse in terms of embedded technology, visibility and mobility, we subdivide them in four categories that help us integrate them in a digital representation of the environment:
• **A resources** are (mobile) interaction devices with an embedded computing platform such as mobile phones and UMPCs. This class of resources is typically used to interact with the environment, but can also host and share software services that give rise to new pervasive applications.

• **B resources** are everyday, often stationary computer-augmented resources such as a smart fridge or a networked light. These resources are part of the physical environment, but also have a digital dimension that can be accessed through **A resources** and by which their state can be observed and possibly manipulated.

• **C resources** are invisibly embedded in the environment, but applications can discover them and read out their state. Typical examples are embedded sensors such as a temperature or motion sensor, whose output is processed by an application.

• **D resources** correspond to physical, non computer-augmented objects that refer to other resources or applications. For example, a pile of DVDs (i.e. a movie collection) could point to a media player, whilst a television panel can act as an abstraction for the set-top box it is attached to.

Most **B**, **C** and **D** resources are known in advance and have a fixed location in the physical environment, while **A** resources are often mobile and not known in advance. Hence we can distinguish between a set of resources that can be integrated in advance in an environment model and a set of dynamic resources that announce their presence at runtime to seamlessly integrate with the pervasive computing environment. We present a tool that enables end-users to create a personalized model of their environment. This model is created beforehand, but continuously adapts to changes that occur when resources enter or leave the environment on the fly.

9.2.1 Integrating places, sights and resources

In Pervasive Maps, an environment is subdivided in different locations (called ‘places’) and sights on a location reveal the resources contained in a particular place. End-users can create digital representations of the environment with the PM editor, as shown in figure 9.3. The modeling process consists of four steps, discussed below.

*Define places.* First, we build up a spatial model of the environment. The end-user identifies different places of interest in the environment which
9.2 Modelling the environment using photos

End-users can create a model of their environment using the PM editor (a). The tool produces a model of the environment relating places, sights, resources and applications (b). Places, sights, resources and applications are hierarchically organized based on their relative locations. For example, a place ‘house’ contains a place ‘first floor’ with places ‘kitchen’, ‘living room’, etc. By subdividing an environment in different places, we can better deal with the environment’s complexity and navigate to resources based on their relative location, i.e. the place they are in. For each place in the environment, an optional floorplan diagram is provided which is created with tools such as Google SketchUp[1]. Floorplan sketches are then imported in the PM editor and annotated with spatial information such as the dimensions of a place and its orientation (e.g. north-east or west).

Define sights. The second step consists of picturing the different places in one’s environment using a digital camera. In particular, images that clearly depict B or D resources embedded in a place are of interest here. We assume camera shots are taken frontally and wider images are preferred over close ups as they generally contain more reference points that help to map the contents of a digital image onto the real world and vice versa.

Define resources. Next, the third step involves integrating available resources into the digital environment. Hereby the end-user is assisted by the

http://sketchup.google.com/
computing system which automatically proposes to integrate $B$ and $C$ resources it discovers on the network. $D$ resources, i.e. resources that are not networked, should be defined manually with a name and icon. Additional attributes describe a resource’s capabilities and technical bindings.

Connect places, sights and resources. Finally, places, sights and resources need to be connected. Places and resources can be marked on imported images. This relates a selected area on the image with a particular place or resource, similar to tagging people on photos that appear on social network sites such as Facebook\footnote{http://www.facebook.com/}. Since photos provide a 2D view on a 3D environment, marked objects might overlap, e.g. when resources occlude each-other. By means of a context menu with ‘bring to front’ and ‘bring to back’ actions, z-orderings are added to tagged objects. Hence we partially reconstruct relevant 3D information that has been lost in the photo. Furthermore, sights are marked as a spot on a place’s floorplan, corresponding to the location where the sight image was taken.

When new resources are integrated in an environment that was already modeled, the model can be altered at runtime by re-applying individual steps. For example, sight photos can be swapped and annotations can be adjusted on the fly. Note also that we can derive extra information from this model that was not explicitly provided by end-users. For example, since a sight provides a view on a place, all resources that were marked on the sight are also located in the place. We can retrieve the relative distances for the resources that are marked on a place’s floorplan. When the user’s position is known, the relative distance between the user and the resources can also be calculated. It is also possible to derive facts from sights such as ‘the microwave is situated left from the fridge’. Moreover, when sights and places are further annotated with spatial information (e.g. compass headings) we can acquire information about the direction in which a resource is located, e.g. in the north-east of the room.

9.2.2 Integrating applications

We define a pervasive application as a front-end for one or more distributed services. In this context, a service is a functional component published on the network, while an application corresponds to an end-user interface that interacts with services in its back-end. The ensemble of a service and its embedded user interface such as a UPnP service with a HTML presentation page can be considered as an application as well. We focus on applications
9.2 Modelling the environment using photos

because users are already accustomed to add, remove and use applications on desktop computers. An application in Pervasive Maps is identified by a URL from where its presentation can be downloaded and has extra properties that specify the type of user interface (e.g. HTML or VoiceXML), version information, etc. The PM editor can discover applications on the network (i.e. service and user interface ensembles) and includes an installer for third-party applications that communicate with services using the PM toolkit (see section 9.4). We use AJAX-based applications since they run on almost every device with a recent Web browser. A distinction is made between private and public applications.

A private application runs in the resource’s firmware and is exclusively used to control the resource it resides on. For example, an application embedded in a kitchen appliance is private, as it can only be used to steer that appliance and e.g. not a similar one which will run its own copy of the application. A shared application runs on any computing device from where it can operate other resources, similar to a word processor that can open and save documents on a network. An example is a light application that takes as input a light resource whose state it can observe and change. Publicly available (Web) applications such as a weather forecast service or an online recipe book can be considered shared applications as well. Private applications are associated with a single resource, while shared applications can operate a range of similar resources or are resource-independent.

We relate resources with applications through meta-data: tags attached to resources and applications are matched at runtime. A positive match indicates that an application can receive a resource as input parameter.

9.2.3 Tag and search

The environment model is further enriched with semantics describing the integrated resources and applications. End-users can attach isa tags to resources that describe the type of the resource, e.g. a fridge, a movie collection, etc. Likewise, applications can be annotated with domain and resource tags, both defined by means of a Tag concept in the PM ontology (see figure 9.4). Domain tags describe the domain of an application (e.g. ‘cooking’, ‘lights’, etc) and are used to categorize applications. Resource tags, on the other hand, help to link resources and applications on the fly. For example, an application with resource tag ‘kitchen appliance’ will be associated with all resources tagged as ‘kitchen appliance’. 
Figure 9.4: Tagging resources and applications using WordNet.

We use the WordNet lexicon to disambiguate between the different meanings a word might have. When tagging a resource or application, the user enters a keyword which is looked up in the WordNet lexicon. The user then selects the intended meaning of the word from a list of word senses which is attached as a tag to the resource. This is realized by relating a Tag instances (PM ontology) with WordSense instances as shown in figure 9.4. The linguistic relations that apply between words can then be exploited to search for resources and applications. For example, an application tagged with the keyword ‘light’ semantically matches a search term ‘lamp’, provided that both keywords are used in the same sense (i.e. a source of illumination). Similar, a keyword ‘piano’ will match a tag ‘musical instrument’ as the former is a hypernym of the latter.

9.3 Exploring the environment

An environment model created using the PM editor is rendered as a Web interface and optimized for mobile devices. This interface provides access to the available applications, either by selecting them from a menu or by browsing through the graph of places, sights and resources. The latter is intended as a spatial variant of the traditional file explorer interfaces.

9.3.1 Application-based navigation

Pervasive applications can be accessed in a traditional way from an alphabetic list of application names and icons. However, since applications dynamically become (un)available, the list of applications can grow long, making navigation difficult. To overcome this, we render a tag cloud from the different domain

3http://wordnet.princeton.edu/
9.3 Exploring the environment

tags an application has assigned, as depicted in figure 9.5. Domain tags automatically categorize applications and allow end-users to locate applications even without knowing their names. This is particularly useful to find applications related to a certain domain such as cooking or to find an application suitable for tasks such as operating the light or adjusting the temperature.

![Image](image1.png)

(a) Searching for an application.

![Image](image2.png)

(b) Recipes application.

Figure 9.5: Applications are organized by their tags. An online recipe book that was integrated in the environment as a shared application is found using a ‘cooking’ domain tag and can be linked with e.g. an oven via a resource tag.

9.3.2 Resource-based navigation

Resources are either listed or shown on top of a map representation. In the list view, places act as folders that hold resources; navigation is menu-based. Opposed to this, the map view is photo-based: users navigate places and sights by selecting interactive parts on floorplan and sight images. For example, a door marked on a photo showing part of the kitchen might point to the living place. Images reveal available, possibly invisibly embedded resources in the environment that can be accessed from anywhere. To highlight resources, we render photos semi-transparent and overlay them with solid pictures of resources. In dense places with many resources, z-orderings help to differentiate between nearby resources and resources further away. However, we acknowledge that we currently do not take into account resources which are completely occluded by other resources; this situation should be avoided when taking a picture, if possible.

Figure 9.6 illustrates the concept of resource-based navigation: different views of the environment help to locate resources and find relevant applications. Note that users can switch between list and map views on the fly; the navigation context is preserved. One can zoom and pan floorplan and sight
Figure 9.6: A combination of list- and photo-based views provide access to the resources embedded in the environment. From left to right, these views show a list of places and resources; an interactive floorplan of a place with a sight reference on it; an interactive sight on which resources are marked; a list of applications associated with a selected resource.
images, making them usable on devices with a limited screen size as well. In the environment depicted in the figure, we attached a resource tag labeled ‘kitchen appliance’ to applications related to the domain of cooking. These applications thus become associated with real-world kitchen appliances such as an oven – tagged as ‘oven’, a hyponym of ‘kitchen appliance’. When navigating to a resource and selecting one of its related applications, the application is loaded with the resource as input so that the application can adapt to the context of use. For example, when opened with an oven resource, the recipes application shown in figure 9.5 could suggest oven dishes by default.

9.3.3 Location and orientation tracking

Context-awareness and in particular location-awareness is considered key for interaction with pervasive environments. Indoor tracking systems are still being investigated extensively in order to improve their accuracy and to reduce costs and setup time. Due to the diversity of real-time positioning techniques (e.g. RFID-based such as LANDMARC [Ni 03] or Wifi-based as discussed in [Chen 09]) we have not considered a default solution, but rather outline how different solutions could be used in conjunction with resource-based navigation in Pervasive Maps. The places and sights that are part of an environment include spatial markers as a special type of meta-information. These markers can be assigned arbitrary data and are added to a place’s floorplan using the PM editor. For example, when used in combination with an RFID-based tracking system, the RFID-tag in the real-world would be matched with a spatial marker in the digital world having the same identifier. Location data is then obtained from the digital marker such as the place it is part of and its relative position w.r.t. this place. Tracking solutions that rely on signal strength (e.g. Wifi-based solutions) can use spatial markers as reference points to store sample data. In this case, a place is overlay with (a grid of) location markers, depending on the tracking system at hand. Algorithms translating real-time positioning data into user-centric location information run as services on a user’s handheld device. Spatial services can leverage the PM toolkit to semi-automatically add spatial markers to an environment model. By walking through the real-world environment with a spatial tag (e.g. a UbiSense tag) and marking the current location on a floorplan using a mobile computing device, end-users can learn the pervasive computing system to become location-aware.

Apart from location, the direction someone is pointed to gives insight into the resource(s) one is facing. This information can be exploited to list just
those resources one is looking at, or to present a sight matching the user’s view in the real-world that reveals the available resources on a digital photo. While creating an environment model, the direction in the form of a compass heading can be assigned to places and sights if applicable (i.e. for non-panorama images). Cameras equipped with a digital compass such as the iPhone 3GS (with integrated 3MP camera and digital compass) can even be programmed to store compass headings automatically in a sight image’s EXIF meta-data.

9.3.4 Augmented reality

An environment modeled using PM can be considered a virtual environment in which resources, sights and places refer to the real-world. By embedding artifacts in the physical environment that refer to the virtual environment, digital and real worlds get intertwined. The different entities in the environment model are all identified by URIs which can be exported to physical tags (e.g. RFID) and attached to physical objects. These tags can then be used for several purposes:

- Locate oneself in an unfamiliar environment, similar to ‘I am here’ information panels. Scanning a tag shows information about the current location of the user, i.e. the place one is currently in.

- Directly access the digital dimension of a resource. Scanning the tag pops up a list of applications related to the resource.

- Create a dedicated ‘start menu’ in the real world composed of tags, icons and textual descriptions. Scanning a tag loads an application.

Augmenting one’s surroundings with digital references contributes to an enhanced interaction experience [Rekimoto 95, Perscha 03]. Although Pervasive Maps does not require an environment to be augmented with digital artifacts, users can still benefit from augmented places, e.g. to avoid navigation in the virtual world’s user interface. In a case study outlined in section 9.5 we have used printed QR codes as physical tags in combination with QR scanning software installed on end-user devices. Unlike an RFID-reader, a digital camera is available on almost any mobile computing device nowadays. QR codes are also easy to generate and can be used without further hardware dependencies.
9.4 Pervasive Maps framework

The architecture of the PM framework consists of four major parts, depicted in figure 9.7:

- **PM host**: The PM host platform is installed on a computer connected to the environment’s internal network, denoted as the PM gateway. It serves a web interface that provides access to the environment’s resources and applications from personal devices.

- **PM client**: A PM client platform is optionally installed on personal devices. It assists end-users in seamlessly accessing a PM environment by taking away the need to start a web browser and manually browse to a PM host.

- **PM editor**: The PM editor is a design and configuration tool for the PM host. It generates a graph representation of the environment from user input, includes an installer for pervasive applications and supports basic user administration.

- **PM toolkit**: Developers of pervasive applications can leverage the PM toolkit to interact with a PM host, e.g. to discover and interact with distributed resources or to query the environment model.

The different parts are further elaborated on next, apart from the PM editor which is already discussed in section 9.2.

![Figure 9.7: Pervasive Maps architecture.](image-url)
9.4.1 PM host

The PM host is a modular platform written in Java with a built-in Web server. The core task of the PM host is to serve a dynamic Web interface for interacting with the environment. This interface is rendered from an environment model provided by the PM editor. It communicates through Javascript and JSON with its Java-based back-end implementation, just like any other deployed Web application. When a resource is opened with an application, the resource object is serialized into a compact JSON variant (known as ‘RISON’) and attached as a parameter to the application’s URL so that an application can quickly validate its input and execute. An authentication component is used to identify the users connected to an environment. Apart from security reasons, the notion of users and the devices they are using is useful information for context-aware applications as outlined in section 2.3.2.

Additional components are also deployed as Web applications on a PM host and expose a REST API and Web interface. A UPnP proxy encapsulated in a Web application announces the PM host as a UPnP control point on the network and translates HTTP requests it receives into UPnP control messages. Furthermore, an integrated query engine enables applications to pose SPARQL queries to interrogate the current environment configuration.

9.4.2 PM client

Personal devices, denoted as A resources according to the classification in section 9.2, fulfill a double role in the environment: they primarily act as a means for interacting with the environment but also offer a computing platform on which services can be installed. To access the PM Web interface from a device, no other software but the device’s native Web browser is required. However, since the IP address of the PM host gateway might be unknown, we install a minimal PM client application on personal computing devices that discovers a PM host and forwards its presentation URL to a Web browser. In addition, PM clients can be extended with plugins to access the device’s hardware/sensors. For example, we extended an iPhone PM client with a plugin to support scanning QR tags with the phone’s built-in camera and processing recognized URIs. Other plugins can be developed to further exploit the interaction capabilities of a device. Although AJAX applications are much more responsive than plain HTML interfaces, they are still constrained by the browser they run in. Native applications, on the other hand, can directly access built-in hardware such as a tilt sensor and as such deliver a richer interaction experience on the device at hand.
9.5 Case studies

9.4.3 PM toolkit

The PM toolkit is a collection of tools for application developers to interface with a PM host and the services it has discovered. These tools basically communicate with the PM’s Web applications using REST APIs for seamless interaction from different programming languages. By default, we support the PM toolkit as a JavaScript library since we mainly target AJAX-based applications. SemSon, discussed in chapter 7, is used to make the concepts defined by the PM domain ontology that populate the environment model (see figure 9.2) accessible from within a Web browser. Using the PM toolkit, developers can easily query the environment model and subscribe to events to receive e.g. notifications when new resources become available. This information can then be exploited to add context-aware behaviour to an application. For example, consider a simple pervasive application for operating the lights in an environment. Via the PM toolkit, the application can discover light resources and locate a service – the back-end of the application – to control lights. A useful feature to integrate in the application’s user interface is the ability to switch on/off all lights in a given place, which is achieved by querying the environment model for those light resources in the selected place and passing them to the light service with the appropriate arguments.

Furthermore, the PM toolkit provides methods to edit the environment model, for instance to add spatial markers and tags. This is particularly useful for configuration tools such as an application that assists the user in setting up a location tracking system which is a pervasive task as well.

9.5 Case studies

As a first evaluation of our system we have implemented two case studies, depicted in figure 9.8. The first case study addresses the scenario of a user visiting an interactive museum. In this scenario, the user explores and interacts with artifacts in the museum, both from within the physical museum as from a remote location. The second case study discusses the design and interaction with a pervasive application to enhance the experience of playing a musical instrument with projected music scores.

9.5.1 Museum visit

We augmented a lab environment with ancient historical objects such as old vases to simulate a single room museum. The Pervasive Maps framework was
deployed on a computer in our museum with two applications installed: one application provides a user interface to control the background music playing in the museum and the other one uses Wikipedia\footnote{http://www.wikipedia.org/} to provide background information about the museum’s artifacts. Using the PM editor, we designed a place ‘museum’ and different resources tagged as ‘artifact’ that correspond to the objects in the museum. As a first experiment within this case study, we asked 8 participants with different backgrounds to take pictures of the museum using a digital camera and to integrate and annotate these as sights in the virtual environment. This included marking the various artifacts on the image and linking them with resources we previously integrated. Next, we asked them to remotely visit the museum over the internet, using their own laptop and preferred Web browser. To guide the visit, we posed a few questions about the artifacts in the museum. The answer to these questions could be found by opening an artifact with the Wikipedia application. Furthermore we asked test users about the background music that was playing in the museum, which they could find out by navigating to an application tagged ‘music’.

In a second experiment, we attached printed QR-codes to the artifacts in the museum to identify them in the digital world. These codes were generated

Figure 9.8: An iPhone running the PM Web interface is used to interact with a museum environment and to select and navigate digital scores.
from the artefact’s URIs using the BeeTagg online code generator\(^5\) which automatically optimizes the size of a URI to fit on a QR-code. We also added QR-tags on the four walls of the museum room which correspond to spatial markers. Using the PM editor, we integrated spatial markers on the room’s floorplan – the spot on the floorplan refers to the location in the room – and added a compass heading to each of them. In ‘map mode’, the spatial information obtained by scanning a spatial marker is used to pan and rotate the place’s floorplan. In ‘sightseeing mode’, the spatial context of the user is used to select the nearest sights the user’s device is pointed to. We asked the same participants of the previous experiment to physically visit the museum using an iPhone with the BeeTagg reader application installed. When a QR-code is scanned, the BeeTagg reader translates the code into a URI and passes it to a Web browser that loads the PM Web interface.

After these experiments we presented the participants with a survey in which they had to indicate their degree of agreement with a number of statements on a five-point Likert scale. The results of the survey are presented in table 9.1. We only included average results in the table since the distributions of responses were all consistent with standard deviation of less than one. Results acknowledge that the PM editor is still a prototype application and needs usability upgrades. Nevertheless, users were able to create a digital museum environment and interact with it through a Web browser. They particularly stressed their interest in a PM user interface for their own places.

### 9.5.2 Digital scores

In this case study we focused on the integration of a dedicated prototype application in Pervasive Maps. We designed a scalable pervasive application for musicians that displays digital scores in the physical environment which assist them with playing their instruments. The setup of the environment is as follows. The Pervasive Maps software is installed on a gateway computer and its Web interface is accessed through an iPhone. A small PC running a Linux-based OS is connected to a projector aimed at the wall above a piano. The computer runs a UPnP service implemented in less than 50 lines of Perl code that exports a control interface for navigating PDF documents. The service leverages XPDF as document renderer and is published on the network using a Perl UPnP DeviceManager implementation\(^6\). Incoming control requests such as `SetDocument` or `NextPage` are dispatched to Perl functions that pass data

\(^5\)http://www.beetagg.com/
\(^6\)http://perlupnp.sourceforge.net/
Table 9.1: Survey statements and average results.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1 It was easy to learn how to use the PM editor.</td>
<td>3.0</td>
</tr>
<tr>
<td>S2 Taking pictures of the museum and annotating them in the PM editor was a positive experience.</td>
<td>3.2</td>
</tr>
<tr>
<td>S3 Interacting with the museum from a Web browser felt intuitive.</td>
<td>3.8</td>
</tr>
<tr>
<td>S4 I had no difficulties in finding more information about artifacts in the museum.</td>
<td>4.0</td>
</tr>
<tr>
<td>S5 I found it easy to find out what kind of music was playing in the museum.</td>
<td>4.1</td>
</tr>
<tr>
<td>S6 A PM interface to observe and control resources in my own house or apartment would be useful.</td>
<td>4.5</td>
</tr>
<tr>
<td>S7 I believe it is simple to map photos on the real world and locate resources that can be interacted with.</td>
<td>3.0</td>
</tr>
<tr>
<td>S8 A combination of digital photos and physically tagged resources improves interaction with the environment.</td>
<td>4.0</td>
</tr>
</tbody>
</table>

to an XPDF process. Furthermore, we created ‘DigiScores’, an AJAX-based Web application that provides access to an archive of score documents indexed on title and composer. DigiScores expects as input a PDF service for rendering scores and a musical instrument to determine the type of scores needed, as piano scores differ from guitar scores for instance. Service and instrument type can be selected from the application’s user interface along with preferred scores as shown in figure 9.8. The set of available PDF services is discovered using the UPnP proxy which is also used to control a service, i.e. to pass a score’s document URI and navigate through its pages. Listing 9.1 shows part of the JavaScript code that initiates the DigiScores application.

```javascript
var r = rison.decode_object(params[‘r’]); // input resource
for (var i=0; i<r.tags.length; ++i) // set instrument type
    if (isSupportedInstrument(r.tags[i].lemma))
        { setInstrument(r.tags[i].lemma); break; }
var e = pm.getEnvironment(); // reference to environment graph
var xpdfs = pm.findServices(‘upnp:xdfl’); // find services
```
for (var i=0; i&lt;pdfs.length; ++i) {
    var d = e.getResourceById(pdfs[i].runsOn);
    var s = new XPDFService(pdfs[i].URL);
    addScoreDevice(d.name,s);
}

Listing 9.1: The DigiScores application initiates by analyzing its input and uses the PM toolkit to discover services.

Next, we modeled a PM environment that consists of a single place, a living room, with a single sight showing a piano which we marked and annotated with a WordNet tag. The tag stipulates the marked area on the sight corresponds to a piano in the sense of keyboard instrument, and not to low loudness which is an alternative meaning of the noun ‘piano’. Next, we installed the DigiScores application via the PM editor and attached it a WordNet tag labeled ‘musical instrument’. Using the iPhone, we can now directly access the DigiScores application via the PM application menu. Alternatively, one can navigate to the piano resource and ‘open it’ with the DigiScores applications whose tag semantically corresponds with the one from the piano. While the DigiScore application does not know about the piano resource, the resource is linked with the current application state, i.e. the selected PDF service, instrument type, last scores, etc, which is automatically cached. By bookmarking the piano as a QR tag and attaching it to the piano in the real world, the DigiScores application becomes directly accessible on the iPhone and will remember the correct settings from a previous session.

9.6 Discussion

In this chapter we have presented Pervasive Maps as an approach to explore and interact with a pervasive environment using mobile devices. Our approach is inspired by desktop operating systems in which a file explorer and a start menu are indispensable tools. We introduce similar tools for interaction with a pervasive computing environment based on a classification of the different types of resources that inhabit a typical environment and the observation that pervasive services are often invisible background processes instead of applications with an end-user interface. We developed a tool that enables end-users to model their own environment by taking pictures and annotating these with extra information. The places and resources identified in the modelling phase can then be explored and interacted with through a Web interface, also from
remote locations. Furthermore, we can improve co-located interaction by embedding tags such as QR codes in the physical environment which refer to resources or applications in the digital environment.

A user study pointed out that the appreciation factor of Pervasive Maps is high, although the PM editor still lacks a satisfactory stage of usability. To improve the modelling tool, users suggested to include a step-by-step wizard for creating a digital version of their environment.
Chapter 10

Conclusions

Contents

10.1 Summary ............................................. 143
10.2 Future research ...................................... 145
10.3 Scientific Contributions and Publications ........ 147

10.1 Summary

In this dissertation we have investigated the development and use of do-it-yourself pervasive applications, allowing end-users to actively participate in their deployment and configuration at runtime. This brought up two main challenges related to the dynamic nature of the pervasive environment these applications are embedded in:

1. coping with dynamic pervasive environments by connecting context information and physical objects so that changes in the digital world are reflected in the real world and vice versa;

2. making users aware of the technology that is surrounding them and providing tools to interact with the digital dimension of a pervasive environment and configure its behaviour.

To address these challenges, we tried to find a good balance between modelling, development and deployment, and interaction techniques to successfully build
and use do-it-yourself pervasive applications. Figure 10.1 provides an overview of the models and frameworks that were developed as part of our research efforts, most of them released under an open-source license. In the remainder of this section we summarize their purpose and indicate how they contribute to reach the above mentioned objectives.

**Figure 10.1:** Relations between models and frameworks that were developed to support do-it-yourself applications.

- **The semantic environment model** uses ontologies to describe the context of resources that populate a pervasive environment. Context-aware applications can acquire context information on demand and get notified of updates on change.

- **The semantic behaviour model** extends the environment model with support for behaviour rules that specify how the system should react when events (i.e. context updates) take place.

- **W2P** is a message-oriented middleware that allows resources to communicate in a pervasive environment by exchanging messages.

- **ReWiRe** offers a middleware platform for deploying service and user interface components that are part of a pervasive application.

- **PerCraft** treats pervasive applications as end-user products that can be (re)deployed via a context-aware setup wizard and observed using pervasive debug tools.
10.2 Future research

- **SemSon** enables the use of ontologies in client-side Web applications and thus brings semantics to the Web browser.

From figure [10.1] we can see that models are extensively used by frameworks at different layers. This denotes the importance of context throughout the lifecycle of a pervasive application: context sensed by real-world resources or served by computational resources is used to guide the deployment of services as well as to decide on a suitable task presentation for a given device. The various arrows leaving ReWiRe in the figure stipulate its central role in the development and deployment of pervasive applications. ReWiRe copes with dynamic environments by maintaining a continuous link between models and software architecture: changes in the configuration of the real world are reflected in the digital world and vice versa. As such, we addresses the first objective with ReWiRe, assisted by SemSon and W2P as semantic Web and communication middleware and the PerCraft design strategy.

Figure [10.1] also depicts that ReWiRe conveniently borrows a meta-UI from the interaction layer, allowing users to locate resources and their supported tasks in a pervasive environment. The meta-UI not only enhances the visibility of the environment, but also integrates tools to monitor the execution of applications and to configure and explain their behaviour. With Pervasive Maps we have further developed the concept of a meta-UI using the most concrete representations one can get of an environment: photos. This accommodates our second objective.

## 10.2 Future research

Pervasive computing is a very broad domain and research is hardly ever finished. In this dissertation we addressed several aspects related to the development and deployment of interactive pervasive applications. The models, frameworks and tools that were created as a result of our research efforts still provide room for various improvements and serve as a source of inspiration for future research.

In part I we have focused on context models whose data is used by frameworks discussed in part II to steer the deployment and behaviour of pervasive applications. Since pervasive applications heavily rely on the network that interconnects their distributed components, it also makes sense to consider quality of service (QoS) properties as a special type of context information. We briefly touched this topic in chapter 6 but we did not investigate techniques to manage the network efficiency such as discussed in [Goeminne 06].
When data collected by QoS algorithms (e.g. network or device statistics) is injected in the environment model, applications could use this information just like any other type of context to adapt to new configurations.

With ReWiRe (chapter 5) and its integrated meta-UI (chapter 8), we provide support for system-driven and user-driven control over the deployment and behaviour of applications. On the one hand, the system can be programmed to autonomously decide on proper reactions when change occurs. On the other hand, the user remains in control at all times assisted by (tools embedded in) the meta-UI. Furthermore, by carrying out user studies, we can get more insight in a proper balance of control between system and users. Pervasive applications can leverage the meta-UI to ask users about how to act in case of uncertainty, resulting in a mixed-initiative approach. Davidyuk et al. [Davidyuk 09] proposed a similar approach for the composition of applications in a ubiquitous environment. We believe wizard services that are considered as the brain of a pervasive application in PerCraft (chapter 6) are particularly suited to engage in a dialog with the end-user when rewiring is needed.

The features that make pervasive computing environments convenient and powerful also make them vulnerable to new security and privacy threats. While our software frameworks are developed with security in mind – W2P (chapter 4) supports secure message channels, behaviour rules are separated from core components in ReWiRe (chapter 5) and users can decide which resources (not) to share – dedicated privacy mechanisms are indispensable to protect context from reaching untrusted resources or prevent services from being invoked by unauthorized parties. Campbell et al. [Campbell 02] propose mechanisms to regulate access control and manage security policies in pervasive computing environments that blend into the background without distracting users too much. Similar to a network layer (W2P) and Web layer (SemSon), a privacy layer could be implemented and adopted by ReWiRe.

In part III we discuss meta-UIs and propose photos as an alternative for menu-based user interfaces to explore a pervasive environment and execute tasks. Photo-based user interface have been studied only little so far, but we believe they have a lot of potential to improve the visibility of resources in the digital environment. For example, as an extension to the Pervasive Maps framework presented in chapter 9 we could overlay state visualizations on a photo such as a yellow glow around a light resource.
10.3 Scientific Contributions and Publications

This section provides an overview of publications that report about the research conducted in this dissertation. These publications were presented at several international scientific conferences. Figure 10.2 depicts how selected publications contribute to this work.

![Diagram of selected publications]

Figure 10.2: Overview of selected publications.


Conclusions


Vanderhulst 09a Geert Vanderhulst, Kris Luyten & Karin Coninx. Photo-based User Interfaces: Picture it, Tag it, Use it. In Proceedings of the 4th International Workshop on Ontology Content (ONTOCONTENT’09), pages 610–615. Springer-Verlag, 2009


10.3 Scientific Contributions and Publications


Appendices
Computersystemen evolueren naar alledaagse omgevingen waarin intelligente toestellen (vaak onzichtbaar) ingebouwd zijn die de gebruiker bijstaan in diens dagelijkse taken. Een dergelijke omgeving wordt aangeduid met de term ‘pervasive’ of ‘ubiquitous’ wat verwijst naar de verscheidenheid aan toestellen die met elkaar verbonden zijn via een (draadloos) netwerk en waarover applicaties typisch gedistribueerd zijn. Als deze applicaties in staat zijn zich aan te passen aan de huidige context (de actuele configuratie van de omgeving), kunnen we tevens spreken van een ‘ambient intelligent’ (AmI) omgeving. Binnen deze thesis bestuderen we de omgang met AmI omgevingen zowel vanuit het perspectief van de software ontwikkelaar als vanuit het standpunt van de eindgebruiker. We stellen twee hoofddoelstellingen voorop:

1. omgaan met dynamische omgevingen door context en fysieke objecten met elkaar te verbinden zodat veranderingen in de digitale wereld gerealiseerd worden in de echte wereld en omgekeerd.

2. gebruikers vertrouwd maken met de technologie die hen omringt en tools voorzien die interactie met de omgeving mogelijk maken evenals het configureren van diens gedrag.

Op een technisch vlak vertaalt dit zich in software die kan omgaan met een dynamische omgeving die wijzigt tijdens de executie van het onderliggende computersysteem. Hiervoor voorzien we in de eerste plaats twee modellen die de huidige context kunnen weergeven: welke apparaten zijn beschikbaar, welke vereisten heeft een applicatie, welke taken kan de gebruiker uitvoeren,
hoe moet het systeem reageren als er een apparaat wegvalt? Beide modellen zijn gebaseerd op ontologien en kunnen ondervraagd worden door applicaties (on demand) en applicaties op de hoogte houden van veranderingen (on change). Het ene model omschrijft zowel algemene concepten (gebruikers, apparaten, diensten, taken, ...), als applicatie-specifieke concepten (lampen, temperatuur, een spel, ...); het andere beschrijft regels die aangeven hoe het systeem zich moet aanpassen aan wijzigingen in de omgevingsconfiguratie (bv. een apparaat dat wegvalt).

Verder stellen we een raamwerk voor dat de distributie van diensten en gebruikersinterfaces ondersteunt binnen een AmI omgeving en deze kan schalen naar een specifieke context of use, voortbouwond op de eerder genoemde modellen. Dit raamwerk maakt gebruik van een zelfontwikkelde netwerkbibliotheek die apparaten en hun toepassingen verbindt over een netwerk en hen toelaat om berichten uit te wisselen, zowel synchroon als asynchroon. Daarnaast bieden we ondersteuning voor Web applicaties die met de opkomst van het Web (ook op mobiele toestellen) niet meer weg te denken zijn. Door dergelijke applicaties toegang te verlenen tot context data, kunnen we ze tevens integreren in een AmI omgeving. Door context te linken met de architectuur van het achterliggende raamwerk, bereiken we een situatie die onze eerste doelstelling vervult: context updates sturen veranderingen aan in de echte wereld en fysieke veranderingen die via software en sensoren aangevoeld worden, zorgen voor context updates in de digitale wereld. De klomtoon ligt hier op het dynamische aspect. Zo zal mogelijk de gebruikersinterface van een toepassing moeten herverteeld worden over de beschikbare apparaten als de context wijzigt en dit op een manier die voor de eindgebruiker(s) acceptabel is. Daarnaast kunnen diensten die aanvankelijk niet ontworpen zijn om samen te werken toch informatie met elkaar delen met als doel de gebruiker beter te ondersteunen in diens taken.

We stellen ons tevens de vraag hoe de software in een AmI omgeving kan geconfigureerd en begrepen worden, wat zowel een behoefte is voor de ontwerper ervan als voor de eindgebruiker. Tools dienen ontwikkeld te worden om te voorzien in een betere zichtbaarheid van applicaties. In het bijzonder richten we ons op tools die een antwoord aanreiken op vragen als “Waarom reageert het systeem op een bepaalde manier en hoe kan dit ongedaan gemaakt worden?” en “Wat zal er gebeuren bij het uitvoeren van een bepaalde actie en hoe kan dit gedrag aangepast worden?”. We stellen voor om dergelijke tools te integreren in een meta-gebruikersinterface die permanent beschikbaar is en dienst doet als een ‘start menu’ voor de omgeving. Een meta-gebruikersinterface presenteert de gebruiker met een overzicht van taken die
kunnen uitgevoerd worden. Op die manier wordt het mogelijk om een AmI omgeving op een intu"tieve manier te verkennen en ermee te interageren via persoonlijke apparaten zoals een mobiele telefoon. We trekken het principe van een meta-gebruikersinterface verder door naar op foto's gebaseerde gebruikersinterfaces. Hierbij worden concrete representaties van een omgeving, namelijk foto's, gehanteerd om door een omgeving te navigeren entaken uit te voeren. Door foto's te annoteren worden interactieve objecten aan systeem en gebruiker kenbaar gemaakt en gelinkt met bepaalde applicaties. Deze meta-gebruikersinterfaces en hun gentegreerde tools bieden een tegemoetkoming aan de tweede doelstelling.
Bibliography


[Vanderhulst 08c] Geert Vanderhulst, Kris Luyten & Karin Coninx. ReWiRe: Designing Reactive Systems for Pervasive En-


