Automating Routine Tasks in Smart Environments

A Context-aware Model-driven Approach

Estefanía Serral Asensio

Thesis supervisors:
Dr. Vicente Pelechano Ferragud
Dr. Pedro Valderas Aranda

Doctoral Thesis

Automating Routine Tasks in Smart Environments

A Context-aware Model-driven Approach

Estefanía Serral Asensio

Supervisors:
Dr. Pedro Valderas Aranda
Dr. Vicente Pelechano Ferragud

Centro de Investigación en Métodos
de Producción de Software

UNIVERSIDAD
POLITECNICA
DE VALENCIA

Automating Routine Tasks in Smart Environments: A Context-aware Model-driven Approach

This report was prepared by
Estefanía Serral Asensio

Supervisors
Dr. Pedro Valderas Aranda
Dr. Vicente Pelechano Ferragud

Members of the Thesis Committee
Dr. Oscar Pastor López, Universidad Politécnica de Valencia
Dr. Joan Fons Cors, Universidad Politécnica de Valencia
Dr. Xavier Franch Gutiérrez, Universitat Politècnica de Catalunya
Dr. Antonio Ruiz Cortés, Universidad de Sevilla
Dr. José Bravo Rodríguez, Universidad de Castilla-La Mancha

Centro de Investigación en Métodos de Producción de Software
Universidad Politécnica de Valencia
Camino de Vera s/n, 46022 Valencia
Spain

Tel:	(+34) 963 877 007 (Ext. 83530)
Fax:	(+34) 963 877 359

Web: http://www.pros.upv.es

Release date: 14-07-2011
Comments: A thesis submitted for the degree of Doctor in Computer Science at the Universidad Politécnica de Valencia.
Title page painting of Vladimir Kush (©Vladimir Kush. All rights reserved).
Rights: ©Estefanía Serral

Acknowledgements

First of all, I would like to thank my supervisors, Dr. Vicente Pelechano and Dr. Pedro Valderas, for always being there and showing me the path to be followed. This work would not have been possible without their valuable guidelines, advice, reviews and discussions. Their expertise, understanding, and patience have contributed a lot to improve this work. Thanks for everything. I also want to express my gratitude to Prof. Óscar Pastor for his direction of our research center, for always having a smile for me and for not exchanging me for camels when he had the chance. Also, I would like to thank the external members of my committee, Dr. Antonio Ruiz, Dr. Xavier Franch and José Bravo for taking time out from their busy schedules and for their valuable comments.

Thanks Javi and Joan, for trusting me and giving me the opportunity of forming part of the pervasive group. Also, thanks for provinding me their support and guidelines in my first steps as a researcher. Thanks Vicky for her catchy laugh and for our amusing conversations. Thanks Miriam, Isma, Nacho and Pablo, for helping me in my work and showing me that the workplace has no reason to be hard, but it may be a place where we can have fun and where people help each other whenever needed. Thanks Salva for his effort, his time and his patience. Also, I want to thank Paqui, Clara, Mario, Maria, Ignacio,

Paco, Nathalie and Sergio S. for helping me in my work by only asking. Thanks Arthur for keeping me company in the long afternoons in the lab. Thanks also to José Luis for his friendship and his incredible sense of humor, his emails made me laugh a lot. Thanks Pau and Carlos, for being my partners, for helping me with the thesis paperwork and, above all, for being a work model to follow. A special gratitude is due to Bea and Giovanni for their friendship, support and encouragement inside and out the lab. I would also like to thank Ana for her kindness and her help with the paperwork, and to the rest of friends and colleagues from the ProS research center for their collaboration.

Outside the work, plenty of people kept me sane and happy; without them, I could not have finished this work. I am grateful to all my friends, among them, Ana and Norma, my friends from Losa and my friends from the university, for their continued moral support and the quality time we spend together.

I want to express my warmest gratitude to all my family, for their constant support, encouragement and love. I know that they want as much as me that I am a Doctor. Above all, I want to thank my parents, Rafa and Amparo, who raised me with love and great values. I am forever indebted to them for their continued understanding, endless patience and encouragement.

Also, I would like to express a special gratitude to my grandmothers. They took care of me the best they could, always providing me with their support and incondicional love. For that, they will always be in my mind and my heart. I wish they still were with me to share my joy with them. Nobody knows how much I miss them.

Finally, I would like to dedicate this thesis to my loving, supportive, and patient boyfriend Guillem. Thanks for being at my side everyday, for trusting me, for putting up with my bad temper, for helping me in whatever I require and for making me laugh whenever I need.

Abstract

Ubiquitous and Pervasive computing put forth a vision where environments are enriched with devices that provide users with services to serve them in their everyday lives. The building of such environments has the final objective of automating tedious routine tasks that users must perform every day.

This automation is a very desirable challenge because it can considerably reduce natural resource consumption and improve users' quality of life by 1) making users' lives more comfortable, efficient, and productive, and 2) helping them to stop worrying and wasting time in performing tasks that need to be done and that they do not enjoy. However, the automation of user tasks is a complicated and delicate matter because it may bother users, interfere in their goals, or even be dangerous. To avoid this, tasks must be automated in a non-intrusive way by attending to users' desires and demands.

This is the main goal of this thesis, that is, to automate the routine tasks that users want the way they want them. To achieve this, we propose two models of a high level of abstraction to specify the routines to be automated. These models provide abstract concepts that facilitate the participation of end-users in the model specification. In addition, these models are designed to be machine-processable and precise-enough to be executable models.

Thus, we provide a software infrastructure that is capable of automating the specified routines by directly interpreting the models at runtime. Therefore, the routines to be automated are only represented in the models. This makes the models the primary means to understand, interact with, and modify the automated routines. This considerably facilitates the evolution of the routines over time to adapt them to changes in user behaviour. Without this adaptation, the automation of the routines may not only become useless for end-users but may also become a burden on them instead of being a help in their daily life. In our approach, the evolution of the automated routines is achieved by simply updating the models. As soon as the models are changed, the changes are also taken into account by the software infrastructure that interprets them to execute the routines. To support this evolution, our approach provides high-level mechanisms as well as a graphical tool that allow the routines to be evolved at runtime by updating the models.

The proposal has been validated by following a case study based evaluaction in which end-users have participated. This validation has proven that our approach is capable of automating the routine tasks that users want to be automated the way they want them to be.

Resumen

La computación ubicua o pervasiva plantea proveer de inteligencia a nuestros entornos desplegando en ellos dispositivos capaces de sensar la información que los rodea (como la temperatura, la localización de los usuarios, etc.) y de controlar los objetos cotidianos (como luces, persianas, etc.) que contienen. Uno de los objetivos más importantes de construir estos entornos es ofrecer servicios a los usuarios que permitan automatizar las tediosas tareas rutinarias que tienen que llevar a cabo cada día.

Esta automatización mejoraría considerablemente la calidad de vida de los usuarios, permitiendo que se despreocupen de realizar estas tareas y evitando que malgasten su tiempo en llevarlas a cabo. Además, al automatizarse, las tareas pueden ser ejecutadas ofreciendo más confort a los usuarios y de manera más eficiente, reduciendo el consumo de recursos naturales como energía y agua.

Sin embargo, la automatización de tareas es un tema complicado ya que, si se realiza erróneamente, puede resultar muy molesto para los usuarios. Si el sistema automatiza tareas que los usuarios no desean o las automatizan de una manera diferente a como ellos quieren, el resultado de la automatización puede acabar siendo intrusivo, interfiriendo en los objetivos de los usuarios, o siendo incluso peligroso. Para evitar estas consecuencias no deseadas, las tareas deben automatizarse atendiendo

a los deseos y necesidades de los usuarios.

Éste es el objetivo principal de esta tesis: automatizar las tareas rutinarias de los usuarios tal y como ellos desean que sean automatizadas. Para conseguirlo, proponemos, por una parte, dos modelos de alto nivel de abstracción para describir las rutinas. Estos modelos proporcionan conceptos cercanos a los usuarios, facilitando su participación en la descripción de las rutinas. Además, los modelos están diseñados para ser interpretados por máquinas, y para proporcionar información suficientemente precisa para poder ser directamente ejecutables.

Por otra parte, proporcionamos también una infraestructura software que es capaz de ejecutar los modelos interpretándolos en runtime; lo que permite automatizar las rutinas tal y como están descritas en los modelos. De esta manera, éstos constituyen la única representación de las rutinas que deben automatizarse, lo que los convierte en los medios principales para entenderlas y modificarlas. Esto facilita considerablemente la evolución de las rutinas para adaptarlas a los cambios de comportamiento de los usuarios. Sin esta adaptación, la automatización acabaría siendo inútil e incluso una molestia para los usuarios. En nuestra aproximación, la evolución de las rutinas se lleva a cabo mediante la adaptación de los modelos, ya que en cuanto éstos cambian, los cambios son tenidos en cuenta por la infraestructura software que los interpreta. Para dar soporte a esta evolución, proporcionamos tanto mecanismos de alto nivel de abstracción como una herramienta gráfica que permite evolucionar las rutinas en runtime mediante la modificación de los modelos.

El trabajo presentado en esta tesis ha sido validado siguiendo una evaluación basada en casos de estudio en la han participado usuarios finales. Esta validación prueba que la propuesta presentada es capaz de automatizar las tareas rutinarias que los usuarios quieren y tal y como ellos quieren.

Resum

La computació ubiqua o pervasiva planteja proveir d'intel·ligència els nostres entorns desplegant en ells dispositius capaços de sensar la informació que els rodeja (tals com la temperatura, la localització dels usuaris, etc.) i de controlar els objectes quotidians (tals com llums, persianes, etc.). Un dels objectius més importants de construir aquests entorns és oferir servicis als usuaris que permeten automatitzar les farragoses tasques rutinàries que han de dur a terme cada dia.

Aquesta automatització milloraria considerablement la qualitat de vida dels usuaris, permetent que es despreocupen de realitzar aquestes tasques i evitant que malbaraten el seu temps en portar-les a terme. A més, en automatitzar-se, les tasques poden ser executades oferint més confort als usuaris i de manera més eficient, reduint el consum de recursos naturals com energia i aigua.

No obstant això, l'automatització de tasques és un tema complicat ja que, si es realitza erròniament, pot resultar molt molest per als usuaris. Si el sistema automatitza tasques que els usuaris no desitgen o les automatitzen d'una manera diferent de com ells volen, el resultat de l'automatització pot acabar sent intrusiu, interferint en els objectius dels usuaris, o sent fins i tot perillós. Per evitar aquestes conseqüències no desitjades, les tasques han d'automatitzar-se atenent els desitjos i necessitats dels usuaris.

Aquest és l'objectiu principal d'aquesta tesi: automatitzar les tasques rutinàries dels usuaris tal i com ells desitgen que siguen automatitzades. Per tal d'aconseguir-ho, proposem, d'una banda, dos models d'alt nivell d'abstracció per descriure les rutines. Aquests models proporcionen conceptes pròxims als usuaris i així faciliten la seua participació en la descripció de les rutines. A més, els models estan dissenyats per ser interpretats per màquines, i per proporcionar informació prou precisa per fer-los directament executables.

D'altra banda, proporcionem també una infraestructura de programari que és capaç d'executar els models interpretant-los en runtime. Això permet automatitzar les rutines tal i com estan descrites en els models. D'aquesta manera, els models constitueixen l'única representació de les rutines que han d'automatitzar-se, la qual cosa els converteix en els mitjans principals per a entendre-les i modificar-les. Això facilita considerablement l'evolució de les rutines per adaptar-les als canvis de comportament dels usuaris. Sense aquesta adaptació, l'automatització acabaria sent inútil i fins i tot una molèstia per als usuaris.

En la nostra aproximació, l'evolució de les rutines es porta a terme per mitjà de l'adaptació dels models, ja que, quan aquests canvien, els canvis són tinguts en compte per la infraestructura de programari que els interpreta. Per tal de donar suport a aquesta evolució, proporcionem tant mecanismes d'alt nivell d'abstracció com una ferramenta gràfica que permet evolucionar les rutines en runtime per mitjà de la modificació dels models.

El treball presentat en aquesta tesi ha sigut validat seguint una avaluació basada en casos d'estudi en la qual han participat usuaris finals. Aquesta validació prova que la proposta presentada és capaç d'automatitzar les tasques rutinàries que els usuaris volen i tal i com ells volen.

Contents

1 Introduction **1**

1.1 Motivation 2

1.2 Problem Statement 6

1.3 Thesis Goals 7

1.4 Design Methodology 9

1.5 Thesis Context 10

1.6 Outline 11

2 Background and Technological Overview **13**

2.1 Ubiquitous Computing vs Pervasive Computing vs Ambient Intelligence 14

2.2 Model Driven Engineering 15

2.2.1 Development Models, Executable Models & Runtime Models 15

2.2.2 Code Generation vs Model Interpretation 17

2.3 Ontology, Ontology Languages and Ontology Reasoners 20

2.3.1 Web Ontology Language (OWL) 22

2.3.2 Pellet: an OWL-DL Reasoner 22

2.3.3 SPARQL 23
2.4 Open Services Gateway initiative (OSGi) 24
2.5 Conclusions 28

3 State of the Art 29
3.1 Analysis Criteria 30
3.2 Machine Learning Approaches 34
3.2.1 Analysis and Discussion 44
3.3 Rule-based Context-aware Approaches 46
3.3.1 Analysis and Discussion 55
3.4 End-user Centred Approaches 56
3.4.1 Analysis and Discussion 65
3.5 Benefits of our Proposal 67
3.6 Discussion and Conclusions 68

4 Overview of the Proposal 71
4.1 Introduction 73
4.2 Process for Automating User Behaviour Patterns 74
4.2.1 SPEM notation 75
4.2.2 The Process Activities 76
4.3 Software Infrastructure 80
4.4 Validation 84
4.5 Conclusions 86

5 Modelling User Behaviour Patterns 87
5.1 Modelling Context 88
5.1.1 The Context Concept 89
5.1.2 Context Modelling in Pervasive Systems 91
5.1.3 An Ontology-based Context Model 92
5.1.4 Tool Support for Creating a Context Model . . . 99

5.2 Modelling the Behaviour Patterns 101
5.2.1 The Task Concept 101
5.2.2 Task Modelling in Software Engineering 102
5.2.3 A Context-adaptive Task Model 105
5.2.4 Tool support 118
5.3 Conclusions 121

6 Automating User Behaviour Patterns 123
6.1 Requirements for Automating Behaviour Patterns . . . 124
6.2 Behaviour Patterns' Automation Process 125
6.3 Software Infrastructure 128
6.3.1 Components of the Software Infrastructure . . . 128
6.3.2 Implementation of the Software Infrastructure . 131
6.4 Conclusions 146

7 Addressing the Evolution of the User Behaviour Patterns 149
7.1 Evolution Characterization 151
7.2 Mechanisms for Evolving the Behaviour Patterns 153
7.3 Tool Support 159
7.3.1 Interface Design Decisions 163
7.3.2 Description of the Graphical User Interfaces . . . 164
7.3.3 Evolving the Behaviour Patterns 171
7.4 Conclusions 174

8 Evaluation of the Approach 175
8.1 Smart Home Case Studies 176
8.1.1 Design of the Smart Home Case Studies 177
8.1.2 Results of the smart home case studies 183
8.1.3 Conclusions of the Smart Home Case Studies' Validation 196

8.2 Nursing Home Case Study 197
8.2.1 Design of the Nursing Home Case Study 197
8.2.2 Results of the the Nursing Home Case Study . . 200
8.2.3 Conclusions of the Nursing Home Case Study Validation 207
8.3 Scalability of Using Models at Runtime 208
8.4 Conclusions 209

9 Conclusions 211
9.1 Contributions 212
9.2 Publications 213
9.2.1 Detail and Relevance of the publications 216
9.3 Future work 217
9.3.1 Combination with Machine-learning Algorithms . 218
9.3.2 Providing Adaptive User Interfaces 219
9.3.3 Interactive and Iterative Tasks and Tasks with State 220
9.3.4 Facilitating the Routine Task Evolution by End-users 222

Bibliography 224

A Software Infrastructure 237
A.1 Model Management Mechanisms Implementation 237
A.1.1 Managing the Context Model: OCean 239
A.1.2 Managing the Task Model: MUTate 242
A.1.3 APIs' Testing 246
A.2 Pervasive Services 248
A.3 Context Monitor 250
A.4 MAtE 253

B Case Study Requirements 257

B.1 Smart Home Requirements 257
B.1.1 An Interview for Identifying the Behaviour Patterns 257
B.1.2 The Identified Behaviour Patterns 259
B.1.3 Required Services 261
B.2 Nursing Home Requirements 263
B.2.1 ACube Requirement Elicitation Artefacts 263
B.2.2 The Identified Behaviour Patterns 269
B.2.3 Required Services 272

List of Figures

1.1 Research methodology followed in this thesis. 10

2.1 The OSGi Service Platform Architecture 25

3.1 Architecture of the Neural Network house project 35
3.2 ISL architecture 38
3.3 Architecture of the MavHome and CASAS projects . . . 41
3.4 An example of HHMM 42
3.5 Architecture of the Henricksen and Indulska's approach 49
3.6 a CAPpella user interface 57
3.7 CAMP user interface 59
3.8 PiP Graphical Interface 64

4.1 Pervasive System Architecture 72
4.2 SPEM notation 76
4.3 SPEM Process for achieving the automation of user behaviour patterns 77
4.4 Software infrastructure 81

5.1 Context ontology classes in ecore format 94
5.2 An example of a context model shown in a tree representation on the top and in OWL code on the bottom 99
5.3 Snapshot of the Protégé user interface 100
5.4 Example of behaviour pattern modelling (graphical representation) 108
5.5 Overview of the task model metamodel 114
5.6 Snapshot of the behaviour pattern modelling tool 120
5.7 Part of the XMI representation of the WakingUp behaviour pattern 120

6.1 Process for Automating User Behaviour Patterns 126
6.2 Automating User Behaviour Patterns 129
6.3 Communication among the components of the software infrastructure 133
6.4 Part of a PervML service model and an example of service code generation 135
6.5 Overview of the OCean API 138
6.6 Overview of the MUTate API 140
6.7 MAtE process for automating the user behaviour patterns 142

7.1 Evolving the executed services using OCean and MUTate 154
7.2 WakingUp model before and after evolving the executed services 155
7.3 Execution traces before and after evolving the executed services 155
7.4 Modifying context conditions using OCean and MUTate 157
7.5 WakingUp model and execution trace after modifying the context conditions 157
7.6 Evolving the service execution plan using OCean and MUTate 158

7.7 WakingUp model and execution trace after evolving the service execution plan 159
7.8 End-user toolkit architecture 160
7.9 Snapshot of the end-user tool for managing the user behaviour patterns 166
7.10 Snapshot of the end-user tool for specifying the context situation of a pattern 167
7.11 Snapshot of the end-user tool for specifying the tasks of a pattern 168
7.12 Snapshot of the end-user tool for managing context information 169
7.13 Snapshot of the end-user tool for managing user information 170
7.14 Evolving the executed services using the end-user tool . 172
7.15 Modifying the context conditions using the end-user tool 173
7.16 Evolving the service execution plan using the end-user tool174

8.1 A context model examples of the smart home case studies 185
8.2 Examples of the models specified in the smart home case studies 186
8.3 JUnit test for evaluating that all the pattern tasks are executed 190
8.4 Results of the PSSUQ Questionnaire 192
8.5 Interface extended with forms for modifying a context condition 194
8.6 Interface extended with forms for modifying a behaviour pattern task 195
8.7 Interface that shows the change validation 196
8.8 Three produced artefacts: a couple of relevant *Personas*, the scenario of *aggressive behaviour* in which they are involved and the slice of correspondent goal model. . . . 201

8.9 Overview of the context model created for the nursing home case study . . . 203
8.10 Specified behaviour patterns in the nursing home case study . . . 205
8.11 Example of a behaviour pattern evolution . . . 207
8.12 Temporal cost of task model operations . . . 209

9.1 Snapshot of an iPhone interface for specifying a behaviour pattern . . . 223

A.1 Class diagram of the context ontology in ecore format . 241
A.2 Overview of the OCean API . . . 243
A.3 Class diagram of the task model metamodel in ecore format 244
A.4 Overview of the MUTate API . . . 246
A.5 JUnit test example . . . 247
A.6 An example of initializePersistentVariables operation that the pervasive services must implement . . . 250
A.7 Code for updating the context model . . . 252
A.8 MAtE process for automating the user behaviour patterns 253
A.9 Code for carrying out the first step of MAtE by using MUTate . . . 254
A.10 Code for executing a system task . . . 256

B.1 Services required for the smart home case studies . . . 262
B.2 Tropos model of the ACube case study. . . . 264
B.3 Transformation performed following the provided guidelines . . . 271
B.4 Services required for the nursing home case studies . . . 273

List of Tables

3.1 Table layout for showing the most important characteristics of each work. X : characteristic not supported or information not published. 33

3.2 Table that summarizes the most important characteristics of NNH. 36

3.3 Table that summarizes the most important characteristics of iDorm. 39

3.4 Table that summarizes the most important characteristics of MavHome and CASAS. 43

3.5 Table that summarizes the most important characteristics of ParcTab. 47

3.6 Table that summarizes the most important characteristics of the Henricksen and Indulska's approach. 50

3.7 Table that summarizes the most important characteristics of the García-Herranz's Approach. 54

3.8 Table that summarizes the most important characteristics of a CAPpella. 58

3.9 Table that summarizes the most important characteristics of CAMP. 60

3.10 Table that summarizes the most important characteristics of Alfred. 63
3.11 Table that summarizes the most important characteristics of PiP. 66
3.12 Table that summarizes the state of the art of the challenges confronted in this thesis. 69

Chapter 1

Introduction

In recent decades, computers have become more and more common in many items such as DVDs, microwave ovens, refrigerators, coffee makers, personal digital assistants, mobile phones, tablets, etc. This proliferation of technology brings Ubiquitous Computing closer to becoming a reality. In Ubiquitous Computing, services are no longer used at the desktop computer, but everywhere to control the items that are used in our daily activities. These services are in charge of functioning invisibly and unobtrusively in the background in order to serve people in their everyday lives and free them to a large extent from tedious routine tasks (Mattern, 2001, 2005).

This is the main goal pursued in this thesis, that is, automating user routine tasks, also known as behaviour patterns. A routine or behaviour pattern is a set of tasks that is characterized by habitual repetition in similar contexts (Neal & Wood, 2007). For instance, some behaviour patterns can be determined by our lifestyle, such as cleaning the house twice a week, or reading electronic mail and opening certain web pages as soon as we have access to Internet; others are reactions to

things happening around us, such as lowering every blind and winding up every awning when it starts to rain, or calling the police when an intruder gets into our home or our store.

The work presented in this thesis deals with automating these behaviour patterns on behalf of users. Until now, several works have been developed to confront this challenge. However, the solutions to this problem still need to be improved upon because of the lack of adequate software methods and models that capture the behaviour patterns according to users' desires and demands, and the difficulty in understanding and evolving the developed systems. In this work, we tackle these problems by proposing a context-aware model-driven approach. It allows behaviour patterns to be described using models of a high level of abstraction according to users' desires and demands. The specified behaviour patterns are automated when needed according to their description by an automation engine that interprets the models at runtime. In this way, all the automations are managed at a high level of abstraction by using the models, which facilitates the understanding and evolution of the automated behaviour patterns. In addition, to support the runtime evolution of the described behaviour patterns, we provide end-users with a tool that provides graphical interfaces to evolve the behaviour patterns.

The rest of this chapter is organized as follows: Section 1.1 explains the motivations for this work. Section 1.2 details the problem that is to solve. Section 1.3 introduces the goals to be achieved. Section 1.4 introduces the research methodology that has been followed. Section 1.5 explains the context in which the work of this thesis is performed. Finally, Section 1.6 gives an overview of the structure of this document.

1.1 Motivation

The more advanced society and technology become, the more interest there is in improving the intelligence of the environments in which we live and work. To achieve this, they are more and more equipped with devices capable of sensing context information and controlling the

state of the objects that surround us. The result is the building of which is known as smart environments. The term smart environment was described in 1991 by Mark Weiser as a *physical world that is richly and invisibly interwoven with sensors, actuators, displays, and computational elements, embedded seamlessly in the everyday objects of our lives, and connected through a continuous network* (Weiser, 1991). Since then, there has been extensive research towards developing smart environments such as digital homes, intelligent work spaces, or hospitals and health care facilities.

One of the most important challenges of building these environments is to automate routine tasks or behaviour patterns. This is because modern life is so busy that time is a premium. Studying, working, doing the housework, socializing, and maintaining a personal life; we must juggle so many tasks that even maintaining a mental list of the tasks to be done becomes difficult. Considering this context, the automation of our behaviour patterns is a very desirable challenge. By automating these patterns, we can not only make our lives more comfortable, efficient and productive, but we can also avoid worrying and wasting time performing tasks that we do not enjoy but that need to be done.

In addition, the automation of our behaviour patterns can also help to deal with an important world challenge: conservation of natural resources, such as energy and water. Despite the advice and guidelines that many Business and NGO's (such as WWF[1], European Climate Foundation[2], etc.) provide us to reduce the consumption of energy and water, it is demonstrated that we use more natural resources each year than we used the previous one. By applying the advice provided by experts on the automation of the behaviour patterns that control lighting, heating and air conditioning, taps, and so on, not only can we optimize water and energy consumption, but we can also save money on our bills.

Nevertheless, although automating user behaviour patterns can bring us great benefits, confronting this challenge is a very complex and

[1] http://www.wwf.org/

[2] http://www.europeanclimate.org/

delicate matter that has not yet been solved. For instance, some of the approaches that have attempted to deal with this challenge are based on machine-learning algorithms. From past user actions, these approaches automatically infer user behaviour patterns and then automate them when the first tasks of an inferred behaviour pattern are detected. For instance, if it is detected that at 8:00 a.m. the user's alarm clock goes off, the lights are switched on, the bathroom heating is switched on, and the coffeemaker makes a cup of coffee, the execution of these tasks will be automatically triggered at 8:00 a.m.

Machine-learning approaches have done excellent work by automatically learning from user behaviour; however, these approaches have some problems. They do not usually take into account users' desires (e.g., the repeated execution of an action does not imply that the user wants this automation). This may make the system automate tasks that users do not necessarily want automated or automate them in a different way from how users want, which may bother users, interfere in their goals, or even be dangerous.

In addition, machine-learning algorithms cannot infer behaviour patterns until they gather sufficient past user actions; therefore, they require a learning period that can take from several weeks to months. Furthermore, they have a lot of difficulty inferring behaviour patterns from several people; thus, they can only learn and automate behaviour for one person. Moreover, since these approaches learn from past user actions, they can only reproduce the actions that users have frequently executed in the past and in the same manner that they were executed. This prevents user tasks from being carried out in a more efficient and comfortable way (e.g., instead of switching on lights, the system could raise blinds when it is a sunny day) and also performing tasks that users did not perform before (e.g., closing windows when users are not at home and it starts to rain).

One way to solve these problems is to tell the system which behaviour patterns the users want to be automated and also how and in which context users want these patterns to be automated. This information is only known by the end-users; therefore, it can only be specified by the end-users or with their participation. The former

option implies the use of end-user tools that allow the end-users to describe their behaviour patterns in an automatable way. To date, several end-user tools have been proposed. However, these tools only provide end-users with limited capacities; therefore, these tools are only appropriate for automating simple tasks such as switching on the lights when user presence is detected. The latter option needs high-level abstractions that allow the behaviour patterns to be captured in an understandable way for users. This would facilitate their participation in the description of the behaviour patterns to be automated. Several works have proposed task models to specify users' tasks in a way that is understandable to them; however, none of the proposed models focus on the automation of user behaviour patterns. Hence, they neither provide enough expressiveness to specify the needed information nor enough accuracy to allow their subsequent automation from their specification. Thus, a technique for properly specifying the behaviour patterns to be automated is still a challenge to be faced.

Furthermore, the specified behaviour patterns must be automated when needed in an unobtrusive way. To achieve this, it is essential to know the up-to-date context on which the behaviour patterns depend. There are already several context-aware approaches that deal with the automation of simple user tasks in the suitable context. These approaches program rules that trigger the sequential execution of actions when a certain context event is produced (e.g., switching on lights when presence is detected). However, although context information is taken into account to automate tasks, these works do not usually take into account the personal desires of each user; therefore, they may still be annoying. For instance, consider that the security system has been programmed to be automatically activated when you leave home. This can be useful because you will not have to do this task anymore, but it can also be a burden if you are absent-minded: you will have to deactivate the alarm each time that you forget something. Furthermore, these techniques are only appropriate for automating relatively simple tasks (Cook & Das, 2005); hence, they usually require large numbers of rules. If we also consider that these rules have to be manually programmed (Cook & Das, 2005), the understanding and

maintenance of the system become very difficult. In spite of the research efforts that have already been done, mechanisms and tools that allow the automation of complex routine tasks when needed and that facilitate understanding and maintenance of the system are still missing.

In addition, the automation of routine tasks is not solved by providing the system with the tasks to be automated and the tools for automating them. This is because users' behaviour may change over time. If the system does not support the evolution of the automated behaviour patterns to adapt to user behaviour changes, the automation of the patterns may become a burden on users instead of a way of helping them. To avoid this, it is essential for the automation of the specified behaviour patterns to be performed in such a way that their evolution is facilitated, i.e., in a maintainable way. Moreover, new tools are required to allow this evolution to be easily performed at runtime without stopping the system (since it has to be performed after system deployment).

1.2 Problem Statement

The automation of user behaviour patterns is not a closed research topic. The above discussion indicates that some problems still need to be considered. The work presented in this thesis attempts to solve these problems, which can be stated by the following three research questions:

Research question 1. How should the behaviour patterns that users want to be automated be represented in order to facilitate that users' desires and demands are taken into account?

Research question 2. How should the specified behaviour patterns be properly automated?

Research question 3. How should the specified behaviour patterns be evolved over time in order to adapt them to new user automation requirements?

These research questions are analyzed and answered in the following section.

1.3 Thesis Goals

The main goal of this thesis is to define a context-aware model-driven approach for improving the automation of user behaviour patterns.

First of all, with regard to **research question 1**, one of the main goals of this work is the specification of the information needed to properly automate the behaviour patterns that users want to be automated. This information is made up of the tasks to be automated for each pattern, and how and in which context these tasks must be automated. To achieve this goal, we propose a context-adaptive task model where each behaviour pattern is specified with user participation as a hierarchical composition of tasks. These tasks are specified according to context in such a way that they are capable of adapting to it. This context information is described in a context model that is based on an ontology. We use an ontology model because according to the works presented in (Baldauf *et al.*, 2007; Chen *et al.*, 2004), it is one of the best ways to describe context. In addition, we use the concept of task not only because it has proved to be effective in user behaviour modelling (Paternò, 2002; Pribeanu *et al.*, 2001; Sousa *et al.*, 2006), but also because it is easily understandable to users (Johnson, 1999). This favours the participation of end-users in their behaviour pattern specification, thereby facilitating that their desires and demands are properly taken into account in this specification.

With regard to **research question 2**, another goal of this work is to automate the behaviour patterns as specified in the models. To achieve this, both the task model and the context model are brought a step further: they are also used at runtime. We develop an infrastructure that allows these models to be interpreted and modified at runtime. This infrastructure is composed by 1) a context monitor that continuously updates the context model according to context changes, and 2) an engine that automates the behaviour patterns according to

context as specified in the models. It is worth noting that rather than translating the models into code, the engine directly interprets them at runtime to automate the patterns as specified. Thus, the behaviour patterns are only represented in the models in such a way that these models are the primary means to understand, interact with, and modify the behaviour patterns that are automated. This allows the behaviour patterns to be easily understood and managed by using concepts of a high level of abstraction instead of code.

With regard to **research question 3**, one of the goals of the present work is to allow the evolution of the automated behaviour patterns to adapt them to changeable user needs. To achieve this goal, we confront two of the most important challenges identified in software evolution: 1) supporting evolution at higher levels of abstraction (e.g., by changing design models) (Ajila & Alam, 2009; Bennett & Rajlich, 2000; Mens, 2009), and 2) supporting post-deployment runtime evolution (Hirschfeld *et al.*, 2004; Mens *et al.*, 2005). Since the behaviour patterns are executed by directly interpreting the models, as soon as the models are changed, the changes are applied in the system. Thus, to confront these evolution challenges, we provide mechanisms that allow the task and context models to be evolved at runtime. These mechanisms use the same high-level concepts used to create the designed models. This allows the evolution of the behaviour patterns to be performed at a high level of abstraction. In addition, we also provide tool support in order to facilitate the adaptation of the behaviour patterns at runtime. This tool provides intuitive graphical interfaces that are inspired by end-user development techniques. The tool reflects the adaptations described in the interfaces by using the designed evolution mechanisms. Thus, adaptations can be performed without the need to stop the system or redeploy it.

Finally, it is worth noting that with this approach, user behaviour patterns can be analyzed in detail before automating them. This achieves a smart environment that can not only automate complex tasks, but one that can also perform them in a more pleasant manner for users and more efficiently in time and energy concerns. Furthermore, it can support the automation of tasks that users want to be automated

although they did not perform them. For instance, using our approach, instead of switching on the bathroom heating at 8:00 a.m. (like machine-learning algorithms would do), the system could switch it on ten minutes before to reach the optimum temperature when the user takes a shower. In addition, the system could wake him with his preferred music instead of the alarm clock going off; and it could also check whether it is a sunny day and, if so, raise the bedroom blinds (instead of switching on the light) to save energy. Moreover, the system could wait until the user enters the kitchen to make coffee so that it was very hot (as the user likes) when the user arrives.

1.4 Design Methodology

In order to perform the work of this thesis, we carried out a research project following the design methodology described by (March & Smith, 1995) and (Vaishnavi & Kuechler, 2004). This design methodology was proposed for performing research in information systems. It involves the analysis of the use and performance of designed artifacts to understand, explain and, very frequently, to improve on the behaviour of aspects of Information Systems (Vaishnavi & Kuechler, 2004).

The design cycle consists of 5 process steps: (1) awareness of the problem, (2) suggestion, (3) development, (4) evaluation, and (5) conclusion. The design cycle is an iterative process; knowledge that is produced in the process by constructing and evaluating new artifacts is used as input for a better awareness of the problem.

Following the cycle defined in the design research methodology, we started with the awareness of the problem (see Fig. 1.1): first, we identified the problem to be resolved, and we stated it clearly. Next, we performed the second step which is suggesting a solution to the problem and comparing the improvements that this solution introduces with already existing solutions. To do this, the most relevant approaches were studied in detail. Once the solution to the problem was described, we planned to develop it (step 3). This step is performed in several phases (see Fig. 1.1). These phases were intended to achieve the proposed

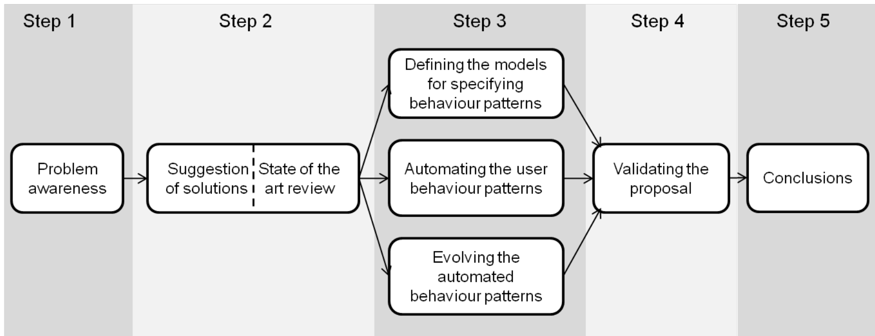

Figure 1.1: Research methodology followed in this thesis.

approach based on models at runtime for automating user behaviour patterns. When the solution was developed, we evaluated the obtained artefacts of the different phases performed in step 3 and validated the whole approach by applying it to several case studies (step 4).

Finally, we analysed the results of our research work in order to obtain several conclusions as well as to delimit areas for further research (step 5).

1.5 Thesis Context

This thesis is being developed in the context of the research center *Centro de Investigación en Métodos de Producción de Software* of the *Universidad Politécnica de Valencia*. The work that has made the development of this thesis possible is in the context of the following research government projects:

- LIFEWARE: Mobilized Lifestyle with Wearables. ITEA 2 project referenced as TSI-020400-2010-100.
- INTERNET DE LAS COSAS COMO SOPORTE A PROCESOS DE NEGOCIO. Project first subsidized by Universidad Politécnica de Valencia with the reference PAID-06-09-2920)

and continued by Generalitat Valenciana with the reference GV/2010/079.

- EVERYWARE: Construcción de Software Adaptativo para la Integración de Personas, Servicios y Cosas usando Modelos en tiempo de Ejecución. CYCIT project referenced as TIN2010-18011.
- SESAMO: Construcción de Servicios Software a partir de Modelos. CYCIT project referenced as TIN2007-62894-AR07.
- OSAMI Commons: Open Source Ambient Intelligence Commons. ITEA 2 project referenced as TSI-020400-2008-114.
- DESTINO: Desarrollo de e-Servicios para la nueva sociedad digital. CYCIT project referenced as TIN2004-03534.
- ATENEA: Arquitectura, Middleware y Herramientas. ProFIT project referenced as FIT-340503-2006-5.

1.6 Outline

The remainder of this work has been structured as follows. First, Chapter 2 explains the technologies and concepts used in this thesis. Chapter 3 compares this work with similar approaches in the area. Chapter 4 gives an overview of the thesis work. Chapter 5 presents the models proposed for specifying the behaviour patterns to be automated at a high level of abstraction. Chapter 6 describes the mechanisms and tools provided for supporting the management of the models at runtime and the automation of the behaviour patterns as described in the models. Chapter 7 presents how the behaviour patterns can be easily evolved over time. Chapter 8 evaluates the presented approach. Finally, Chapter 9 summarizes the results of this work and describes the future work.

Chapter 2

Background and Technological Overview

Research in Pervasive Computing is very diverse since the field itself has not yet been clearly defined. Researchers from different communities make efforts to understand and improve concepts, technologies and applications for research in Pervasive Computing. In order to clarify the foundations on which our approach relies, the concepts, techniques and technologies used in this thesis are introduced in this chapter. The remainder of the chapter is structured as follows: Section 2.1 describes the different terms used to refer to pervasive computing. Section 2.4 explains the OSGi technology used for implementing the systems developed by the presented approach. Section 2.2 defines the term Model-Driven Engineering (MDE) and the different MDE concepts and techniques for understanding this thesis. Section 2.3 explains what is an ontology and the main ontology languages and tools used in this work. Finally, Section 6.4 concludes the chapter.

2.1 Ubiquitous Computing vs Pervasive Computing vs Ambient Intelligence

The main goal of this thesis is providing a context-aware model-driven approach for achieving the automation of user tasks in smart environments. A smart environment is a "physical world that is richly and invisibly interwoven with sensors, actuators, displays, and computational elements, embedded seamlessly in the everyday objects of our lives, and connected through a continuous network" (Weiser, 1991). In this context, several terms are used in the published literature for talking about similar concepts. The main differences depend on the context of use: Academy vs Industry and EEUU vs Europe.

On the one hand, Mark Weiser coined the term ubiquitous computing in a more academic and idealistic sense (Mattern, 2001, 2005). He saw it as omnipresent services that serve people in their everyday lives at home and at work, functioning invisibly and unobtrusively in the background and freeing people to a large extent from tedious routine tasks. On the other hand, industry (IBM) coined the term pervasive computing, with a slightly different slant (Jochen Burkhardt & Rindtor, 2002; Uwe Hansmann & Stober, 2001), as an explosion of interconnected "smart devices", from watches to cars, that can make user lives easier and more productive. Here, investigation is focus on what forms these devices might take, what new functions they might perform, and ways to pack more computing ability into smaller spaces.

In (Bohn *et al.*, 2005; David Wright & Punie, 2005) is stated that while researchers in the United States were working on the vision of ubiquitous computing, the European Union began promoting a similar vision for its research and development agenda. The term adopted in Europe is ambient intelligence (coined by Emile Aarts of Philips) as "a vision where people will be surrounded by intelligent and intuitive interfaces embedded in everyday objects around us and an environment recognizing and responding to the presence of individuals in an invisible way". This point of view is confirmed by the great number of events and research projects that are organized and/or funded in Europe under this

term whose topics clearly matches the ones that are inside the scope of the ubiquitous computing area.

Although subtle differentiations could be done between these terms according to their etymological meanings (neither ubiquitous implies intelligence, nor intelligence implies pervasiveness, etc.), we can state in general that the main idea or vision behind them is the same and, therefore, they can be equally used in this thesis.

2.2 Model Driven Engineering

The work presented in this thesis applies the guidelines provided by Model Driven Engineering (MDE). It aims to raise the level of abstraction in program specification and increase automation in program development. The idea promoted by MDE is to use models at different levels of abstraction for developing systems, thereby raising the level of abstraction in program specification. The major advantage of this is that models can be expressed using concepts that are much less bound to the underlying implementation technology and are much closer to the problem domain. This makes the models easier to specify, understand, and maintain (Selic, 2003). The increase in automation of program development is reached by transforming the higher-level models into lower level models until they can be executable using either code generation or model interpretation.

We next define the terms of MDE that are needed for understanding the approach proposed in this thesis.

2.2.1 Development Models, Executable Models & Runtime Models

In MDE, a model is an abstraction or reduced representation of a system. When the models are used in the software development phase, they are named **development models**. Development models are models of levels of abstraction above the code level that are used for representing some aspect of the software to be developed. For

instance, technology-independent models of software describe systems using concepts independent of the underlying computing technologies.

If development models are fully expressive to be automatically executed, they are considered **executable models**. Executable models can be executed by means of translating them into the system code that will be executed (such as in the works presented in (Mellor & Balcer, 2002; Muñoz *et al.*, April 2006)) or by using an interpreter/engine that directly executes what is specified (such as in the work presented in (M.B. Juric & Sarang, 2006)). Further discussion about these two ways of executing a model can be found in the next subsection. When the models are used at runtime, they are named **runtime models**. Runtime models are models that present views of some aspect of an executing system and are therefore abstractions of runtime phenomena (France & Rumpe, 2007). Runtime models can provide a richer semantic base for runtime decision-making in order to achieve system adaptation, since they provide up-to-date and exact information about the runtime system. Thus, the system itself can query the models at runtime in order to make adaptation decisions, to choose the adaptation strategy, and to control and to steer the adaptation process.

It is also important to note that development models may be used as runtime models if they are still used at runtime, and runtime models as development models if they are used to evolve software systems. In this last case, a runtime model can be seen as a live development model that enables dynamic evolution.

In (Blair *et al.*, 2009) a very interesting classification of models was stated according to the following characteristics:

- Structural or behavioural: Models specify either the structure of the system or its behaviour aspects. Structural (or architectural) models specify system components and their connections; objects, inheritance relationships and invocation pathways, etc. In contrast, behavioural models specify the functionalities provided, how the system is going to react to the events, etc.

- Procedural or declarative: Procedural models specify the real structure or behaviour of the system; while declarative models

specify this in terms, for instance, of system goals.

- Functional or non-functional: Functional models specify the system functionality, while non-functional models are used for specifying non-functional requirements as security, temporal and memory cost, etc.

- Formal or informal: Formal models are inspired by the mathematics of computation; whereas informal models are derived from consideration of programming models or domain abstractions.

2.2.2 Code Generation vs Model Interpretation

In MDE, Code Generation and Model Interpretation are two different strategies to make executable models become a reality. Both strategies are used in practice.

Code generation is used to generate the code of an application from a higher level model that describes it. To generate the code in existing programming languages and platforms from the model, a model compiler (many times defined as a template engine using languages of model-to-text transformation such as Mofscript, Xpand, etc.) is implemented.

In case of model interpretation, code is not generated to create the code of the application. Instead of this, a generic engine is implemented (for instance in Java) to directly interpret and execute the model.

Next we explain some of the most important advantages (many of them discussed in the Code Generation conference of 2010) of these approaches compared to each other:

Advantages of Code Generation

Code generation has the following advantages in comparison to model interpretation:

- It protects intellectual property: with code generation an application can be generated and delivered to a specific client. With model interpretation, the runtime engine, which allows a whole class of applications to be implemented, has to be given to the client as well as the application.

- Easier to start with: if multiple applications for the same domain have been built, code generation can be used. The code being the same for all applications (i.e., static code) can be put in a domain framework, and the code that is specific for each application (i.e., variable code) can be generated by creating a Domain-Specific Language to model the variability and using templates that transform it into the variable code.

- It provides an additional check by the compiler: when code is generated, that code needs to be compiled. Thus, compilers also check the generated code for errors. In case of an interpreter, either these checks have to be done during the interpretation of the model or a tight coupling between the modelling environment and its interpreter has to be created.

- Changes in templates are easier to track: code generation templates are just text files, hence changes can be easy to track (e.g., by using a version control system). The same holds for changes in the code of the interpreter, however, this code is generic and its less clear what exactly has changed.

Advantages of Model Interpretation

Model interpretation has the following advantages in comparison to code generation:

- It enables faster changes: changes in the model do not require an explicit regeneration, rebuild, retest, and redeploy step. This will lead to a significant shortening of the turnaround time.

- It enables changes at runtime: because the model is available at runtime it is even possible to change the model without stopping the running application.

- Easier to change for portability: in principle, an interpreter creates a platform independent target to execute the model. It is easy to create an interpreter which runs on multiple platforms. In case of code generation you need to make sure you generate code compliant to the platform. In case of model interpretation, the interpreter is a black box, it does not matter how it is implemented as long as it can run on the target platform.

- Easier to deploy: when code generation is used, the generated code has to be often opened in Eclipse (Eclipse, 2011) or Visual Studio [1] and built to create the final application. In case of model interpretation, the interpreter just has to be started and the model has to put into it. Code is not necessary any more.

- Easier to update and scale: it is easier to change the interpreter and restart it with the same model. You do not have to generate the code again using an updated generator. The same can hold for scaling: scaling an application means initializing more instances of the interpreter, executing the same model.

- It is more secure: The interpreter provides an additional layer on top of the infrastructure, everything underneath is abstracted away. This is essentially the idea of a Platform-as-a-Service (PaaS).

- It is more flexible than code generation: there are limits to template-based code generation. In these cases, helper files to extend the possibilities of template based code generation are needed. An interpreter can be less complex in these cases, and often less code is needed to accomplish the same result.

- Debug models at runtime: while the model is available at runtime, it is possible to debug the models by stepping through them at

[1] http://msdn.microsoft.com/es-es/vstudio/aa718325

runtime. This only holds for action languages, not for declarative languages. When debugging at model level is possible, domain experts can debug their own models and adapt the functional behaviour of an application based on this debugging. This can be very helpful when complex process or state models are used.

2.3 Ontology, Ontology Languages and Ontology Reasoners

In philosophy, Ontology is the study of being or existence and its basic categories and relationships. It seeks to determine what entities can be said to ”exist”, and how these entities can be grouped according to similarities and differences. We have used ontologies for millennia to understand and explain our rationale and environment. However, only recently ontologies have become a research topic of interest in computer and information science.

In computer and information science, an ontology is a formal representation of a set of concepts within a domain and the relationships between those concepts. It is used to reason about the properties of that domain, and may be used to define the domain. In philosophy, ontologies have a main goal: to establish the truth about reality by finding an answer to the question “what exists”. In the world of information systems, in contrast, an ontology is a software artefact (or formal language) designed with a specific set of uses and computational environments in mind. An ontology is often something that is ordered by a specific client in a specific context and in relation to specific practical needs and resources. In a widely-quoted definition, an ontology is “a specification of a conceptualization” (Gruber, 1993). Thus, an ontology allows a programmer to specify in an open and meaningful way the concepts and relationships that collectively characterize some domain.

An ontology mainly contains the following elements:

- Classes: all kinds of existences or concepts. A class usually refers

to a collection or a category of objects sharing some common character and well accepted under common sense.

- Attributes: properties that identify a class itself from other classes.

- Relationships: A relation between two ontology classes interprets how the two classes, more precisely the objects of these classes, are related. Typically a relation is a particular connection between two classes that specifies how an object is connected to the other in an ontology.

- Individuals: instances or objects of the defined classes. All the objects under a category are named as "individuals" of this class.

The terms Abox and Tbox are also used to refer to the elements of an ontology. These terms describe two different types of statements in ontologies. Tbox statements describe a system in terms of controlled vocabularies, for example, a set of classes and properties. Abox are Tbox-compliant statements about that vocabulary. Tbox statements are sometimes associated with object-oriented classes and Abox statements are associated with instances of those classes. Abox and Tbox statements together make up a knowledge base (a special kind of database for knowledge management).

Thus, an ontology is an explicit, first-class description. This description can be specified in different languages, such as RDF or OWL, and it can be used by different reasoners, such as Racer (Haarslev & Möller, 2003) or Pellet (Sirin *et al.*, 2007). This is one of the main reasons for building a context ontology: we can use a reasoner to derive additional truths about the concepts that we are modelling. Next, we introduce OWL, which is the ontology language used in this thesis, Pellet, which is the reasoner used, and SPARQL, which is the query language used for reasoning.

2.3.1 Web Ontology Language (OWL)

Web Ontology Language (OWL) (Smith *et al.*, 2004) is a semantic markup language for publishing and sharing ontologies on the World Wide Web. In our approach, we have selected OWL for implementing an ontology-based context model.

The OWL Web Ontology Language is designed for being used by applications that need to process the content of information instead of just presenting information to humans. OWL facilitates greater machine interpretability of Web content than that supported by XML, RDF, or RDF Schema (RDF-S), by providing additional vocabulary along with a formal semantics. OWL has three increasingly-expressive sublanguages: OWL Lite, OWL DL, and OWL Full. OWL DL does not permit some constructions allowed in OWL Full, and OWL Lite has all the constraints of OWL DL plus some more. The intent for OWL Lite and OWL DL is to make the task of reasoning with expressions more tractable. Specifically, OWL DL is intended to be able to be processed efficiently by a description logic reasoner. OWL Lite is intended to be amenable to processing by a variety of reasonably simple inference algorithms. We have selected OWL DL because of several reasons:

- It enables automated reasoning.
- It has the capability of supporting semantic interoperability to exchange and share context knowledge between different systems, i.e., contexts can be exchanged and understood between different systems in various domains.
- It is also more expressive than other ontology languages such as RDF.
- It is an open World Wide Web Consortium (W3C) standard.

2.3.2 Pellet: an OWL-DL Reasoner

In this approach, we use Pellet (Sirin *et al.*, 2007) to derive additional truths about the modelled context information. Pellet is a complete

and capable OWL-DL reasoner with very good performance, extensive middleware, and a number of unique features. It is written in Java and is open source under a liberal license. It is used in a number of projects, from pure research to industrial settings.

Pellet is the first implementation of the full decision procedure for OWL-DL (including instances) and has extensive support for reasoning with individuals (including conjunctive query over assertions), user-defined data types, and debugging ontologies. It implements several extensions to OWL-DL including a combination formalism for OWL-DL ontologies, a non-monotonic operator, and preliminary support for OWL/Rule hybrid reasoning. It has proven to be a reliable tool for working with OWL-DL ontologies and experimenting with OWL extensions.

2.3.3 SPARQL

SPARQL (SPARQL, 2010) is a recursive acronym that stands for SPARQL Protocol and RDF Query Language. As the name implies, SPARQL is a general term for both a protocol and a query language. In this thesis, we use the SPARQL acronym to refer to the query language.

SPARQL was standardized by the RDF Data Access Working Group (DAWG) of the W3C, and is considered a key semantic web technology. On 15 January 2008, SPARQL became an official W3C Recommendation. This query language is based on graph-matching techniques. Given a data source, a query consists of a pattern which is matched against the data source, and the values obtained from this matching are processed to give the answer. The data source to be queried can be an OWL model as the context model proposed in this context.

A SPARQL query consists of three parts:

- The output selection part: a SPARQL query can be a yes/no query (ASK), a selection of values of the variables which match the patterns (SELECT), a creation of new triples (INSERT), and a description of resources (DESCRIBE).

- The pattern matching part: it includes several features of pattern matching of graphs, like optional parts, union of patterns, nesting, filtering (or restricting) values of possible matchings, and the possibility of choosing the data source to be matched by a pattern.
- The solution modifiers part: once the output of the pattern has been computed in the form of a table of values of variables, this part allows to modify these values applying classical operators like projection, distinct, order, limit, and offset.

2.4 Open Services Gateway initiative (OSGi)

The implementation of the approach proposed in this thesis uses an OSGi server for running the developed systems. The Open Services Gateway initiative (OSGi), more known now as the OSGi Alliance (OSGI, 2011) is an independent, non-profit corporation founded in 1999. Since then, the OSGi Alliance has been working to define and promote open specifications for the delivery of services to networks in homes, cars, and other environments. These specifications enable a development model where applications are dynamically composed of many different reusable components, which are known in the OSGi terminology as bundles.

The OSGi Service Platform provides functionalities to Java that makes Java the premier environment for software development. Java provides the portability that is required to support products on many different platforms; while the OSGi technology provides the standardized primitives that allow applications to be dynamically constructed from reusable and collaborative bundles.

To achieve this, the OSGi Service Platform allows the composition of bundles to be changed dynamically without requiring restarts. To minimize the coupling, as well as make these couplings managed, the OSGi technology provides a service-oriented architecture that enables these components to dynamically discover each other for collaboration. The OSGi Alliance has already developed many standard component interfaces for common functions like HTTP servers, configuration,

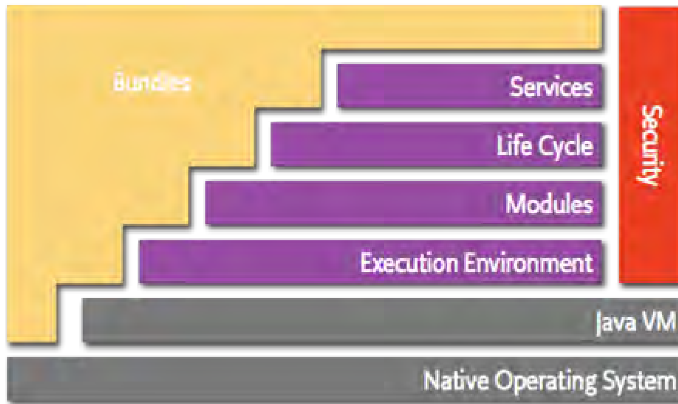

Figure 2.1: The OSGi Service Platform Architecture

logging, security, user administration, XML and many more.

The architecture of the OSGi Service Platform is shown in Figure 2.1. The core component of this architecture is the OSGi Framework, which is divided in four layers:

L0-Execution Environment: is the specification of the Java environment.

L1-Modules: defines the class loading policies which establish how a bundle can import and export code.

L2-Life Cycle Management: it provides the API for installing, starting, stopping, updating and uninstalling bundles at runtime.

L3-Service Registry: provides a comprehensive model to share objects between bundles. To do this, it connects bundles in a dynamic way by offering a publish-find-bind model for plain Java objects.

As shown, the bundles are installed over this framework, which runs over a Java Virtual Machine (JVM).

In addition, the OSGi Service Platform provides constructs and services that offer many important benefits for developing context-aware pervasive systems such as the following ones:

- Device Discovery. OSGi relies on device discovery using low-level protocols as EIB, Lonworks or UPnP. When the devices are discovered they can be coupled to device drivers and then used for the system services.
- Adaptation. Adaptation is achieved through dynamic bundle loading and updating, and service lookup. When a new device or service is registered in the framework by a bundle, any other running service can use it. The link is done at runtime.
- Easy Deployment. The OSGi technology specifies how components are installed and managed by defining an API. This standardized management API makes it very easy to integrate OSGi technology in existing and future systems.
- Small size. The OSGi Release 4 Framework is implemented in about a 300KB JAR file. This is a small overhead for the amount of functionality that is added to an application by including OSGi. OSGi therefore runs on a large range of devices. It only asks for a minimal Java VM to run and adds very little on top of it.
- Fast. One of the primary responsibilities of the OSGi framework is loading the classes from bundles. In traditional Java, the JARs are completely visible and placed on a linear list. Searching a class requires searching through this list. In contrast, OSGi pre-wires bundles and knows for each bundle exactly which bundle provides the class. This lack of searching is a significant speed up factor at startup.
- Integration. The integration of the software representation of a device and the physical environment relies on low-level technologies. Basically, OSGi uses bridges to the final device drivers. The native device drivers are in charge of the communication with the physical device.

- Programming Framework. OSGi provides a well defined programming framework around the service concept that separates service description from any possible implementation. In addition, as OSGi is Java-based, it is operative system independent. For complex applications, there is a proposal and implementation of a component model built on top of OSGi.

- Robustness. Dynamic coupling of services and devices is a guarantee of robustness. If a service runs out or a device fails they can be automatically replaced by other elements that provide the same functionality.

- Security. The framework security model is based on the Java 2 specification. OSGi defines a standard service for permission administration. In the framework, a bundle can have a single set of permissions. These permissions are used to verify that a bundle is authorized to execute privileged code. For example, a FilePermission defines what files can be used and in what way.

- Widely Used. The OSGi specifications started out in the embedded home automation market, but since 1999 they have been extensively used in many industries: automotive, mobile telephony, industrial automation, gateways and routers, private branch exchanges, fixed line telephony, and many more. Since 2003, the highly popular Eclipse Integrated Development Environment runs on OSGi technology and provides extensive support for bundle development. In the last few years, OSGi has been taken up by the enterprise developers. Eclipse developers discovered the power of OSGi technology but also the Spring Framework helped popularize this technology by creating a specific extension for OSGi. Today, OSGi technology can be found at the foundation of IBM Websphere, SpringSource Application Server, Oracle (formerly BEA) Weblogic, Sun's GlassFish, and Redhat's JBoss.

- Supported by Key Companies. OSGi counts on some of the largest computing companies from a diverse set of industries such as:

Oracle, IBM, Samsung, Nokia, IONA, Motorola, NTT, Siemens, Hitachi, Deutsche Telekom, Redhat, Ericsson, etc.

2.5 Conclusions

The purpose of this chapter was to provide a brief introduction to the existing background on top of which this work is built on. We have explained the different concepts needed for understand the presented work and the paradigms where it has been developed as well as the techniques and technologies used for developing it and that will be applied in the following chapters.

Chapter 3

State of the Art

The automation of behaviour patterns is the automatic execution of tasks that users perform everyday. This is a complicate and delicate matter because tasks must be executed in a non-intrusive way and attending users' desires and demands; otherwise, the automation of behaviour patterns may bother users, interfere in their goals or even be dangerous. This implies executing tasks on behalf of users when they need and the way they want them.

This chapter revisits and analyses the most popular and relevant approaches found in the literature that deal with the challenge of intelligently acting on behalf of users. In order to classify these approaches, we use the taxonomy published in (Chin *et al.*, 2008), which defines three different categories: machine-learning approaches, rule-based context-aware approaches and user-centred approaches.

Machine-learning approaches cover those based on the use of machine learning mechanisms. These approaches automatically derive a prediction model of future user behaviour from observation of the past user actions.

Rule-based context-aware approaches cover those in which developers or manufacturers program context-aware rules for automating user behaviour. These approaches are focus on the management on context, and the developed rules are usually composed of two parts: context conditions and tasks that must be triggered when these conditions are fulfilled.

User-centred approaches are those commonly referred to end-user programming. They are characterised by the use of techniques that allow non-technical people to create "programs" to customise the functionality of their own environments.

Next, we first suggest a set of important dimensions to classify and analyse the approaches found in the literature. Next tree sections describe the approaches related with our work placed on each one of the described categories, providing analysis and discussion of them at the end of each section. After this analysis, we explain the most important benefits of our approach. Finally, we present general discussions and conclusions of the related work.

3.1 Analysis Criteria

This section explains the dimensions used to analyse the related work. These dimensions characterize how the challenges confronted in this thesis are managed. Specifically, according to the challenge confronted, the following dimensions are studied:

- Regarding the modelling of behaviour patterns at design time:
  - Type of model: it indicates the type of the model/s used for specifying the automations of the system.
  - Expressivity: it indicates the expressivity of the model/s used in terms of support for specifying conditions, context-awareness, temporal relationships between the tasks to be automated (or only sequence of actions), and hierarchy task descriptions.

- Support for user participation: it indicates whether or not the approach facilitates user participation in the specification of the models. To avoid that the automation of the behaviour patterns may be intrusive or non-desirable, they should be specified according to users desires and demands. To achieve this, user participation is required.
- Supported by tools: it indicates if tools are provided for facilitating the specification of the model/s.

• Regarding the implementation of behaviour patterns:

- Technique: it describes the technique or method used for implementing the automations. For instance, manual, using algorithms, code generation, model interpretation, etc.
- Implemented by: it indicates if the implementation of the behaviour patterns is automatic or manual; and in the later case, who implements the automation (only developers, only end-users, or both).
- Supported by tools: it indicates if tools are provided for developing the automations.

• Regarding the automation of behaviour patterns:

- Capabilities: it indicates what capabilities are provided for automating user tasks, i.e., only sequence of tasks or also temporal relationships between the tasks to be automated, context-adaptivity and context reactivity of the tasks, task abstraction hierarchies, etc.
- Dynamic representation of the automations: it indicates how the automations are stored/managed at runtime.
- Control: it indicates if the automations are only controlled by the system (the system decides what is going to be automated and when) or by the end-users (the end-users keeps the control about what have to be automated and in which conditions) or by both (the system decides what is going to be automated and when but the end-users can modify

these decisions by specifying, for instance, their preferences). Much of the behavioural literature on information system acceptance (Heijden, 2003) suggests that users actually prefer to stay in control over their systems.

  - Scalability: it indicates if the automation approach is prepared to be scalable, which is a critical problem in pervasive computing (Satyanarayanan, 2001).

- Regarding the evolution of the automated behaviour patterns (in accordance to the taxonomies published in (Buckley *et al.*, 2003; Lientz & Swanson, 1980)):

  - Level of support: it indicates what can be evolved and when and how is evolved.
  - Developers vs end-users: it indicates who can perform the evolution (developers, end-users or both).
  - Level of abstraction: it indicates where and in which level of abstraction the evolution can be carried out; i.e., at a low level of abstraction by changing code, at a high level of abstraction by changing the models (at modelling level).

This information is summed up for each work using the layout of Table 3.1. As well as the above explained dimensions, this table also shows the following important characteristics in the automation of user behaviour:

- Application domain: in which domains the developed applications can be applied and whether they are independent of the domain or specific of some domain.

- Number of users: number of users for which automations can be performed; i.e., if the approach only supports the automation of actions for only one user or if it supports the automation of actions for many users.

Finally, the table also summarizes the specific limitations of each work.

Approach		
Modelling	Type of model	
	Expressivity	
	User Participation	
	Supported by tools	
Implementation	Technique / Method	
	Developers vs end-users	
	Supported by tools	
Automation	Capabilities	
	Dynamic representation of the automations	
	Scalability	
	Control (end-users vs system)	
Runtime Evolution	Level of support	
	Developers vs end-users	
	Level of abstraction	
Domain		
Number of Users		
Limitations		

X: not supported or not published information

Table 3.1: Table layout for showing the most important characteristics of each work. X : characteristic not supported or information not published.

Paying more attention in the properties above explained, we next describe the most important approaches related with our work according to the identified categories.

3.2 Machine Learning Approaches

Machine Learning Approaches use machine-learning algorithms capable of predicting or inferring user behaviour from past user actions and then automating this inferred behaviour according to the past observations. For instance, let's suppose that the user switched on the bedroom lights at 8:00 a.m., then switched on the heating of the bathroom, and afterwards made a coffee. If the algorithm detects that this sequence is usually performed, the algorithm classifies it as a behaviour pattern and then automatically triggers the execution of this sequence of actions at 8:00 a.m. Next, we present some relevant examples of this type of approach.

The Neural Network house. The Neural Network house (NNH) (Mozer, 1998) was proposed by Mozer. It uses a neural network for achieving two different goals: anticipate the user's needs and minimize energy consumption by automatically controlling light, heating, water and ventilation. To achieve this, Mozer et al. developed an adaptive control of home environments' (ACHE) system.

Its architecture (which is shown in Figure 3.1) is structured in four layers that are replicated for each control domain (lighting, air heating, water heating, and ventilation). At the lowest level, the state of the environment and models about the occupancy of the rooms are obtained by monitoring the environment. Using this context information, predictors based on artificial neural networks, forecast future states (such as expected hot water usage, expected occupancy patterns, etc.). These future states are then passed to setpoint generators, which are in charge of determining a setpoint profile specifying the optimal value for each environmental variable (e.g. light or air temperature) over a window of time. Finally, the setpoint profile is passed to the device regulators, which are in charge of controlling the physical devices to achieve the optimal value using the minimum set of actions (e.g. increase lamp1's intensity 2 points and lower lamp2

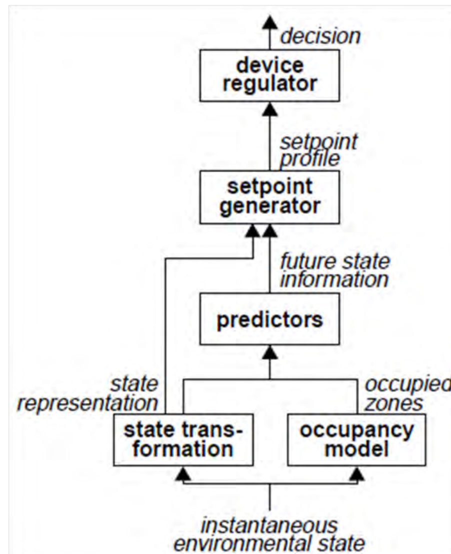

Figure 3.1: Architecture of the Neural Network house project

and lamp3 3 points each).

The setpoint generator requires knowledge about inhabitant preferences, while the device regulator has knowledge about the physical layout and characteristics of the environment and controlled devices. In this way, if the inhabitants or their preferences change over time, only the setpoint generator need to relearn. Setpoint generators and device regulators are based on one of two approaches to control: indirect control using dynamic programming and models of the environment and inhabitant, or reinforcement learning.

A neural network is a powerful model for inferring patterns, however, it is a complex mathematical model of a low level of abstraction and very difficult to understand (i.e., the system cannot explain its reasoning process to users in an understandable manner). In addition, both establishing the overall goals and deciding how they are accomplished are system decisions,

completely avoiding user control. Furthermore, although NNH supports learning from multiple users, it restricts the automation to just certain environmental variables in the house domain.

Table 3.2 summarizes the relevant information of NNH according to the presented layout.

NNH		
Modelling	Type of model	X
	Expressivity	X
	User Participation	X
	Supported by tools	X
Implementation	Technique / Method	Learning by using neural networks
	Developers vs end-users	Automatic
	Supported by tools	Yes
Automation	Capabilities	sequence of actions to achieve a goal according to context
	Dynamic representation of the automations	Neural network
	Scalability	X
	Control (end-users vs system)	System
Runtime Evolution	Level of support	X
	Developers vs end-users	X
	Level of abstraction	X
Domain		Home
Number of Users		Multiple
Limitations		- Relationships between tasks not suported, and automations limited to lighting, air heating, water heating, and ventilation. - The control is totally in the system. - Difficult understanding and maintenance

Table 3.2: Table that summarizes the most important characteristics of NNH.

The iDorm project. The iDorm (Hagras *et al.*, 2004) project is a test bed for ubiquitous computing environments. It predicts user behaviour by learning fuzzy rules that map sensor state to

actuator readings representing inhabitant action. A fuzzy rule is defined as a conditional statement in the form:

IF x is A

THEN y is B

In the iDorm project, x and y are context variables managed by the devices of the environment, and A and B are values of these variables. These rules are stored in a text file, but the authors have implemented a small parsing tool to convert this text file into a human-readable format.

The iDorm contains areas for varied activities such as sleeping, working, and entertaining. To make the iDorm as responsive as possible to its occupant's needs, it has a set of embedded sensors (such as temperature, occupancy, humidity, and light-level sensors) and effectors (such as door actuators, heaters, and blinds). All of them are connected to a network infrastructure and some of them contain agents.

In addition, a physically static computational artefact closely associated with the building is also connected to the network. This artefact contains the iDorm embedded agent. This agent receives sensor values through the network, contains the user's learned behaviour, and computes the appropriate control actions using the fuzzy-logic-based Incremental Synchronous Learning (fuzzy ISL) system. It then sends them to iDorm effectors across the network.

Figure 3.2 shows the ISL architecture, which forms the learning engine in the iDorm embedded agent. The ISL works as follows: when new users enter the room, they are identified by the active key button, and the ISL enters an initial monitoring mode where it monitors the inputs and the user's action and tries to infer rules from the user's behaviour. Learning is based on negative reinforcement because users will usually request a change to the environment when they are dissatisfied with it. After the monitoring period, the ISL enters a control mode in which it uses the rules learned during the monitoring mode to guide its control

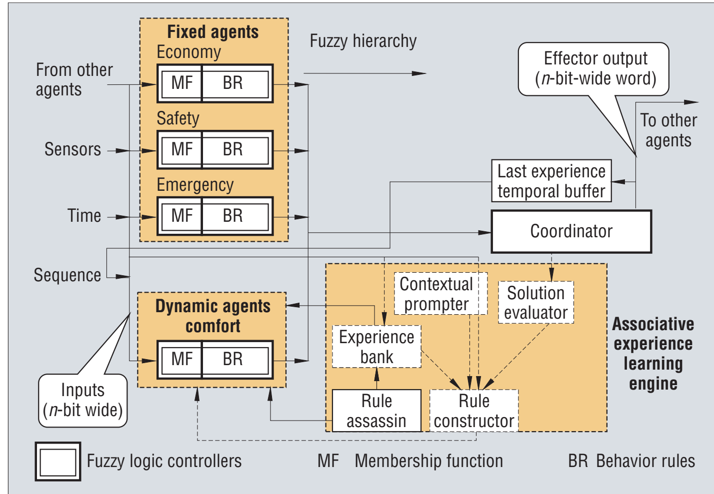

Figure 3.2: ISL architecture

of the room's effectors. Whenever the user behaviour changes, it might need to modify, add, or delete some of the rules in the rule base. Thus, the ISL goes back to the monitoring mode to infer rule-based changes by determining the user's preferences in relation to the specific components of the rules that have failed. All the consequents of the rules that were unsatisfactory to the user are changed. Up to date, iDorm supports behaviour learning from one user, but it is prepared for learning behaviour for more users if it is done in an isolated way.

As well as the learned behaviour, iDorm also contains fixed behaviours that are preprogrammed. The fixed behaviours are predefined because they cannot be easily learned (they include safety, emergency, and economy behaviours).

Since embedded agents have limited computational and memory capabilities, the number of stored rules is limited to 450. Each rule will have a measure of importance according to how frequently it is used. When the system reaches the memory limit, the

iDorm		
Modelling	Type of model	X
	Expressivity	X
	User Participation	X
	Supported by tools	X
Implementation	Technique / Method	Automatic learning of fuzzy rules that are located in agents
	Implemented by	Automatic
	Supported by tools	Rules are automatically learned by the fuzzy-logic-based Incremental Synchronous Learning (fuzzy ISL) system
Automation	Capabilities	Fuzzy rules: Event-action rules
	Dynamic representation of the automations	Text file
	Scalability	Low, they are limited to 450 rules
	Control (end-users vs system)	System
Runtime Evolution	Level of support	Rules are automatically added, modified, and deleted as necessary
	Developers vs end-users	Automatic
	Level of abstraction	Low
Domain		Home
Number of Users		Only one, but prepared for learning behaviour for more users in an isolated way.
Limitations		- Not support for relationships between tasks. - The control is totally in the system. - Low scalability, difficult understanding and maintenance.

Table 3.3: Table that summarizes the most important characteristics of iDorm.

Rule Assassin retains rules according to the priority of highest frequency of use.

The iDorm project have done excellent work in the automation of user actions by predicting them from user behaviour. Although the implemented system needs some period to be able to predict these actions, it introduces an advantage regarding the other

machine-learning approaches, which is implementing a fixed behaviour that is executed from the very beginning. However, the project still keeps the control in the system and does not consider user desires for the automation of their actions. Furthermore, the presented approach neither provide support for establishing context relationships among the actions to be automated nor configuring these actions by using context information. In addition, the rules that they learned are very simple (like event-action rules), therefore, they need a great amount of rules to meet user needs. This factor and the limited computational and memory capabilities of their agents make the scalability of the approach very difficult.

Table 3.3 summarizes the relevant information of IDorm according to the presented layout.

MavHome and CASAS. MavHome (Cook *et al.*, 2003; Youngblood *et al.*, 2005) and CASAS (Rashidi & Cook, 2009) are two projects directed by Diane J. Cook in the School of Electrical Engineering and Computer Science at Washington State University. CASAS is the continuation of the MavHome Project.

These projects assume that people are creatures of habit; therefore, MavHome and CASAS apply automatic user behaviour learning through observation.

Their architecture, as shown in Figure 3.3, is designed with modular components and has four cooperating layers:

- The Physical layer contains the hardware available in the house. This includes all physical components such as sensors, actuators, network equipment, and computers.
- The Communication layer routes communications between the users and the house and the house and external resources. It is available to all layers to facilitate communication and service discovery between components.
- The Information layer gathers, stores, and generates knowledge useful for decision making.

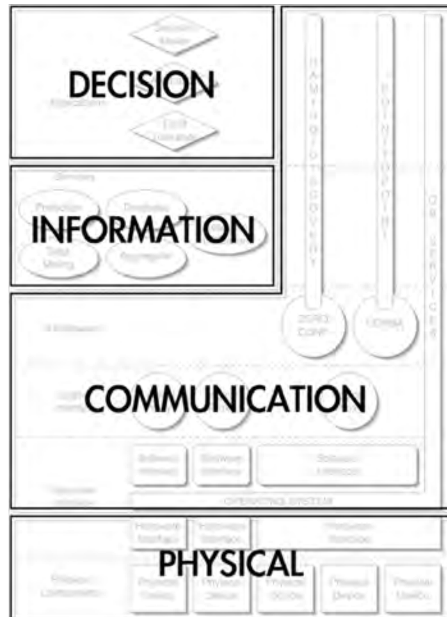

Figure 3.3: Architecture of the MavHome and CASAS projects

- The Decision layer learns from stored information and makes decisions on actions to automate in the environment. In addition, this layer also provides adaptation by altering the transition probabilities between events based on feedback in order to improve automation performance.

Using this architecture, MavHome performs the following process: sensors monitor the environment and make information available through the communication layer to information layer components. The database stores this information while the data-mining algorithm encapsulates these observations in event-based chains of a Hierarchical Hidden Markov Model (HHMM) forming behaviour patterns with an exact periodicity (see an example of HHMM in Figure 3.4). This HHMM is extended by tying actions and rewards to the transitions between states

forming a Hierarchical Partially-observable Markov Decision Process (HPOMDP) model. The observation data and data-mined patterns are also used to train an episode membership algorithm. This algorithm is used by the decision-maker layer to try to locate where in the HPOMDP model the inhabitant's activities are currently engaged. Once successful, the decision-maker looks ahead in the model to determine events that will occur in the near future, and if these events are within the control of the system it can issue actions to automate them. The decision actions are communicated to the information layer which records the action and communicates it to the physical layer. The physical layer performs the actions changing the state of the system.

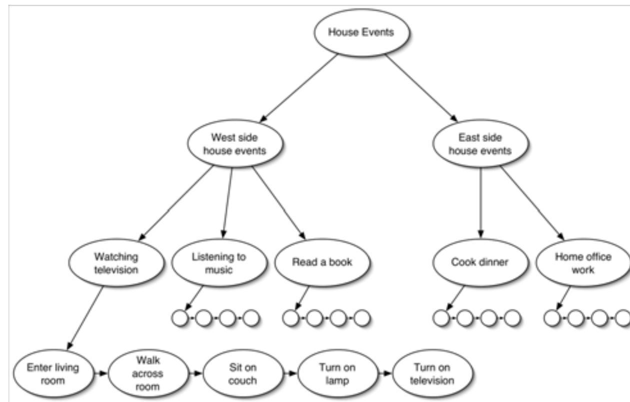

Figure 3.4: An example of HHMM

CASAS uses the same architecture but provides other algorithms that allow behaviour patterns to be predicted with more information such as duration and start times. In addition, CASAS also provides an end-user tool that allows the user to adapt the inferred behaviour patterns. For instance, using this tool, end-users can add activities, delete activities, or modify entire activities by adding, deleting, or reordering the events that comprise the activity. The user can also indicate that the adaptation is automatically performed.

MavHome & CASAS		
Modelling	Type of model	X
	Expressivity	X
	User Participation	X
	Supported by tools	X
Implementation	Technique or Method	Automatic user behaviour learning through observation
	Performed by	Automatic
	Supported by tools	Yes
Automation	Capabilities	Sequence of actions with temporal relations
	Dynamic representation of the automations	Hierarchical Hidden Markov Model
	Scalability	X
	Control (users vs system)	Both
Runtime Evolution	Level of support	Adding, modifying and deleting behaviour patterns
	Developers vs end-users	End-users and automatic
	Level of abstraction	Low
Domain		Home
Number of Users	Only one vs multiple users	Only one
Limitations		- Not support for context adaptation or context relationships among tasks. - Do not provide information about how runtime adaptation is applied in the system

Table 3.4: Table that summarizes the most important characteristics of MavHome and CASAS.

Both MavHome and CASAS are outstanding examples of machine-learning approaches. Although MavHome kept the control in the system, CASAS extends this to allow the user modify this behaviour if so desired. However, they still automates user behaviour without considering user desires, what may be intrusive for them. In addition, they do not use context

information to properly automate user actions, instead, they only based on the occurred events. Thus, they neither provide support for establishing context relationships among the actions to be automated nor configuring these actions by using context information. Furthermore, these projects are focused on the home domain and are restricted to only one inhabitant in the home.

Table 3.4 summarizes the relevant information of this approach according to the presented layout.

3.2.1 Analysis and Discussion

The presented approaches have obtained great results in automating user behaviour patterns inferring them from the actions performed by users using machine-learning algorithms. However, the automatic learning is a difficult task that has several important limitations.

First, most of the presented algorithms have the cold-start problem: they cannot start to predict until they have captured enough past actions, which may takes weeks or even months. For this reason, these approaches are usually applied in the smart home domain, since it is a domain where routine tasks are more often performed. If we consider routines that are performed once a week (e.g. only on Saturdays) or only in certain circumstances (e.g., fertilize the land according to the season), the training may take even longer. In our approach, the application domain is not limited to smart homes and a learning process after system deployment is not required. All the behaviour patterns that are known at design time and that users want to be automated are automated from the very beginning.

In addition, machine-learning algorithms act on the basis of what happened, according to what they see happening and believe is going to happen, but without considering users' desires nor knowing users' goals. This lack of knowledge may lead to automating tasks which the user may not want automated (users may not want to automate everything they do) or reach generalizations in such a way that the automation becomes a burden on the users instead of a way of helping

them. Another related problem that automatic learning presents is that users lose the control of the system. It is the system who makes the decision of what automating and when, which may cause anxiety on users (Heijden, 2003). Contrary to the majority, CASAS allows end-users to change the automations. However, these automations can only be changed after been executed, which may have already bothered users because they want to change them. These two problems can cause the loss of user acceptance of the system

In this thesis, we attempt that users are involved in all the process. Analysts analyse users' tasks to identify behaviour patterns that can be automated and specify them. Thus, analysts can use their knowledge and their experience to improve the performance of the identified behaviour patterns, but always taking into account users' desires because users participate in this specification. In addition, users can also evolve the automations to adapt them to their needs; therefore, the system only automates the tasks that users want to be automated (users are always in control of the system), facilitating user acceptance.

Another important problem is that machine-learning algorithms are based on the performed user actions; therefore, they can only reproduce these actions and usually as a sequence of actions, without considering conditions (according to context or temporal restrictions) among the tasks. In addition, users cannot do all they would like to automate (e.g. turn on/off all the lights at the same time). In our approach, behaviour patterns to be automated are specified using users' knowledge and desires and analysts' knowledge and experience; therefore, any behaviour pattern can be automated even though users did not perform it before.

Finally, machine-learning approaches represent the automations at a low level of abstraction. Thus, if evolution is supported, it is also done at a low level of abstraction, e.g., by changing states or rules. Unlike these approaches, we propose to represent the automations at a high level of abstraction by using high-level concepts (such as task, user, location, etc.); thus, runtime evolution can be performed at modelling level using these concepts.

3.3 Rule-based Context-aware Approaches

Rule-based context-aware approaches implement context-aware rules for automating user actions (e.g. switching lights off when there is nobody in a room). These rules are programmed by developers or manufacturers and hard-coded in the system. Next, we present some important examples of this type of approach.

ParcTab project. ParcTab at Xerox Parc is one of the pioneering research projects centred in Ubiquitous Computing. It defines the PARCTAB system (Want *et al.*, 1992), a prototype developed to explore the impact and possibilities of mobile computation in an office environment. The system is based on three types of devices of different sizes: tabs, pads and boards; and it is composed of three types of software components: gateways, agents, and applications. Gateways implement a service for sending and receiving packets using IR signals. Each tab is represented by an agent, which tracks the location of its tab and provides location independent reliable remote procedure calls. Applications are built using a library of widgets designed to accommodate the PARCTAB's low IR-communication bandwidth and small display area. A distinguished application, the "shell", permits a tab user to start and switch among applications.

Over this system, different context-aware applications are programmed. Of special interest for this work are the context-triggered actions (Want *et al.*, 1995) of these applications. Context-triggered actions are simple IF-THEN rules used to specify how context-aware systems should be adapted. To do this, a rule specifies an action that should be executed in a certain context. As an example, the watchdog program monitors Active Badge activity and executes Unix shell commands in response. A user configuration file (containing a description of Active Badge events and actions) is loaded on start-up. Entries of the configuration file, codifying the IF-THEN rule, are of the form:

The PARCTAB system		
Modelling	Type of model	X
	Expressivity	X
	User Participation	X
	Supported by tools	X
Implementation	Technique / Method	Manually by using a library of widgets
	Developers vs end-users	Developers
	Supported by tools	No
Automation	Capabilities	IF-THEN rules
	Dynamic representation of the automations	user configuration file
	Scalability	X
	Control (end-users vs system)	system
Runtime Evolution	Level of support	X
	Developers vs end-users	X
	Level of abstraction	X
Domain		Office environment
Number of Users		Multiple, but without dealing with conflicts
Limitations		- Manual implementation - Not support for context adaptation or relationships between tasks - The control is in the system - Not support for the runtime evolution of the rules.

Table 3.5: Table that summarizes the most important characteristics of ParcTab.

badge location event-type action

where badge and location are strings matching the badge wearer and current location, event-type is a badge event type (i.e., arriving, departing, settled-in, missing, or attention) and action is a Unix shell command.

Even though the PARCTAB system is one of the pioneers in the research of Ubiquitous Computing, this system already tackles one of the most fundamental problems of Smart Environment automation: their multiple population. It allows different users

to have different preferences over the same objects; however, no mechanism for coordinating conflicting preferences is supplied. In addition, the simplicity of the language does not have the necessary flexibility to address complex problems: its triggers were fixed to a badge, location and event type.

Table 3.5 summarizes the relevant information of this approach according to the presented layout.

Henricksen and Indulska's approach. Henricksen et al. present in (Henricksen & Indulska, 2006; Henricksen *et al.*, 2006) a set of models to specify and support the development of context-aware approaches that provided reactive behaviour to context changes. Specifically, the authors provide a graphical Context Modelling Language (CML) as an extension of Object-Role Modeling (ORM). CML is focused on supporting the specification of context-aware communication applications. Specifically, the model captures the following: user activities, associations between users and communication channels and devices, and locations of users and devices. The authors also provide a procedure for mapping from CML to a relational representation.

In order to define context conditions based on CML, the authors propose the situation abstraction. They define a *situation* as a set of logical expressions formed using context variables. Each logical expression combines any number of basic expressions using the logical connectives *and*, *or* and *not*. In our approach, we base on this abstraction to define context conditions since it provides a great expressivity to form them.

In addition, they propose a preference model that supports the ranking of choices according to context. Each preference is a pair consisting of a scope and a scoring expression. The scope describes the context situation in which the preference applies. The scoring expression assigns a score to a choice. Preferences can be grouped into sets and combined according to policies. A policy has a single score that reflects the score of all preferences in the set. The policies dictate the weights attached to individual

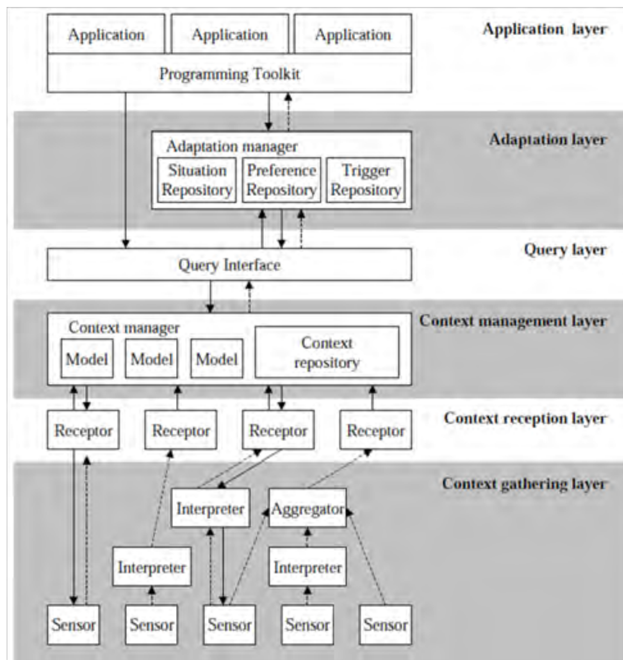

Figure 3.5: Architecture of the Henricksen and Indulska's approach

preferences and determine how conflicting preferences are handled. These preferences are also stored in a relational model.

As well as these design models, the authors proposed two programming models: the branching model, which assists in decision problems involving a context-dependent choice among a set of alternatives; and the triggering model, which describes event-condition-action rules to support event-driven programming. In the triggering model, the precondition and the event are specified in terms of situations, and the actions are specified using a programming language as Java.

To support these models, the authors provide a software infrastructure, which is shown in Figure 3.5, that is organized into the following layers:

Henricksen & Indulska's approach		
Modelling	Type of model	Graphical context model and a preference model
	Expressivity	Context information
	User Participation	No
	Supported by tools	No
Implementation	Technique / Method	Manually event-driven programming
	Developers vs end-users	Developers
	Supported by tools	They provide a software infrastructure to facilitate application development
Automation	Capabilities	ECA rules adaptive to context
	Dynamic representation of the automations	Context and preferences: relational model Triggers: Hard-coded rules
	Scalability	X
	Control (end-users vs system)	System, but end-users can change their preferences to configure the rules
Runtime Evolution	Level of support	X
	Developers vs end-users	X
	Level of abstraction	X
Domain		Communication applications
Number of Users		Multiple
Limitations		- Not support for relationships between tasks - Manual implementation - Not support for the evolution of the rules over time.

Table 3.6: Table that summarizes the most important characteristics of the Henricksen and Indulska's approach.

- The context reception layer translates context inputs into fact-based representation that uses the context gathering and management layers.
- The context management layer is responsible for maintaining a set of context models and instantiations of them.
- The query layer provides applications and the preference management layer with a convenient interface with which

to query the context management infrastructure. The query layer also implements a transaction model that allows synchronous queries to be performed against a set of context information.

- The adaptation layer is responsible for storing repositories of preferences and evaluating preferences using the services of the query layer.
- The application layer provides a Java API to support the branching model. This API provides a variety of methods for evaluating and selecting one or more candidate triggers according to the context.

This approach introduces very useful abstractions and concepts, such as situation and preference, for automating user actions. However, although this approach provides a software infrastructure that facilitate application development, these applications have to be still manually implemented. In addition, it does not allow establishing relationships among the tasks to be automated. Also, the approach does not support the evolution of the implemented rules; however, it allows end-users to configure these rules by changing their preferences, but tools are not provided for supporting this configuration.

Table 3.6 summarizes the relevant information of this approach according to the presented layout.

García-Herranz's Approach. The approach proposed by García-Herranz et al (García-Herranz *et al.*, 2010) presents a working solution to end-user programmable context-aware smart homes. He designed a rule-based language in which users' preferences can be codified by the end-users as reaction rules in the form of Event Condition Action (ECA) rules. This language allows context-dependent composite events through the use of timers and using an event logic as well as expressing mixed consumption policies. The language is an analogy of the natural "When *trigger* if *conditions* then *actions*" structure, where:

- Triggers are supervised context variables responsible for activating the rule. Only disjunction of primitive events is allowed in the triggers part.
- Conditions are a set of “context variable-value” pairs representing a context state that needs to be satisfied for detonating the action. Only conjunction of conditions is allowed. Disjunction can be codified as separate disjunction-free rules.
- Actions are “context variable-value” pairs to be set when, given a triggered action, all its conditions evaluate to true.

These parts are structured according to the following template:

```
trigger1 || trigger2 || ...
  ::
    condition1 && condition2 && ...
  =>
    action1 && action2 && ...
;
```

For instance, to specify that the alarm is turned on if the main door is opened for 5 minutes, the following rule is created:

```
door:main_door:status ::
 door:main_door:status = 1
=>
  TIMER 5m 1
  {   device:alarm:status := 1 ;  }
  {   door:main_door:status ::
      door:main_door:status = 0
    =>
    TIMER.kill
   ;
  }
;
```

These ECA rules are internally indexed in three hash tables that index the rules through their triggers, conditions and actions.

The author states that this language can deal with most of the requirements derived from having an end-user as a programmer such as application-independent programming, an increasing degree of complexity, a simple and flexible mechanism of expression, means of explanation, or automatic learning.

In addition, he proposes an agent-based programming structure in which the rules and the rule engines are distributed. Each agent has its own set of rules and is related to the user or group of users that created it, as well as to the elements they affect with their rules. This programming structure helps to deal with other end-user requirements such as maintaining the user's trust in the system when a part of it fails, solving conflicts. To deal with multiple users, they allow the creation of hierarchies of users in such a way that when a conflict is found, the triggers that are executed are those associated to the user that is higher in the hierarchy.

The goal of this work is not to develop any particular interface but, conversely, to create a programming mechanism that can be used through many interfaces. The work provides a language for creating automation rules that is thought to be understood by end-users, therefore, it lacks expressivity to specify context-aware relationships among tasks. In order to facilitate rule creation, the author has developed a basic GUI that is not an end-user interface, but it facilitates the process for individuals with low programming skills by abstracting users from the grammar of the agent rule mechanism. In addition, they have also developed an initial prototype for an end-user interface based on the Magnet Poetry metaphor proposed by the CAMP approach, which will be next explained.

Table 3.7 summarizes the relevant information of this approach according to the presented layout.

Garcia-Herranz's Approach		
Modelling	Type of model	X
	Expressivity	X
	User Participation	X
	Supported by tools	X
Implementation	Technique / Method	Manually programming the rules using an Agent-based programming structure
	Developers vs end-users	End-users or developers
	Supported by tools	Yes, graphic interfaces
Automation	Capabilities	ECA rules with timers: context-aware conditions, sequence of actions, timers between tasks.
	Dynamic representation of the automations	Three hash tables that index the rules through their triggers, conditions and actions
	Scalability	Distributed implementation: rules and the rule engines are distributed among agents.
	Control (end-users vs system)	End-users
Runtime Evolution	Level of support	X
	Developers vs end-users	X
	Level of abstraction	X
Domain		Smart homes
Number of Users	Only one vs multiple users	Multiple
Limitations		- Not support for context-aware relationships or task hierarchy. - It may be tedious for end-users to specify the system - The control is in the system - Not support for the evolution of the rules over time.

Table 3.7: Table that summarizes the most important characteristics of the García-Herranz's Approach.

3.3.1 Analysis and Discussion

Context-aware rule-based approaches have made great advances in introducing context into software systems. To automate user tasks, they program rules that trigger the sequential execution of actions when a certain context event is produced. However, they do not provide context conditions among the tasks to be automated; therefore, they require large numbers of manually programmed rules (Cook & Das, 2005). This makes these approaches not appropriate for automating user complex tasks and difficult to maintain. In contrast, in our approach, all the routine tasks that the system automates are described and managed by using a task model, which provides a great expressivity for specifying complex tasks.

In rule-based approaches, rules are manually implemented by developers or manufacturers, thus, although most of these approaches provide software infrastructure for facilitating the development of automated system, they have to be still manually implemented. In addition, rules are fixed hard-coded in the appliances before they are supplied to users; therefore, they do not usually take into account users' desires. An exception is the Garcia-Herranz's approach, which also provides an interface for allowing end-users to specify the rules. However, specifying rules from scratch may be tedious for the end-users. In our approach, we propose executable models of a high level of abstraction to specify automations, therefore, their implementation is done by only specifying these models. In addition, in our proposal, end-users participate in the design of these models and can evolve them at runtime when needed; therefore, users' desires are always taken into account.

The evolution of rules is another problem that present the approaches placed in this category: whilst the developed automated systems offer many automated features (e.g., switching lights off when there is nobody in a room), they do not allow people to alter the rules.

3.4 End-user Centred Approaches

End-user centred approaches provide alternatives for end-users to program their environments. The vast majority of these approaches are focused on end-user programming by presenting particular UIs and languages. Next, we present some relevant examples of this type of approach.

a CAPpella. *a CAPpella* (Dey *et al.*, 2004) is a Context-Aware Prototyping environment intended for end-users to build applications without writing any code. To achieve this, this prototyping environment uses the programming by demonstration technique: the user shows the program its desired context-aware behaviour (situation and an associated action) in situ. a CAPpella uses a combination of machine learning and user input for recording this behaviour.

To achieve this, a CAPpella has 4 main components:

- A recording system, which provides multimodal sensing capability to capture both the situation and the action that should be taken.
- An event detection, which is the process of deriving higher-level events from the raw data produced by the sensors.
- A user interface, which is shown in Figure 3.6. It consists of two main panels, an event panel for viewing the captured events and a player panel for watching and listening to the captured audio and video. In this interface, users select the events and also the streams of information that they believe relevant to the demonstrated behaviour.
- A machine learning system, which is the system for testing or training from the events selected in the interface. In either case, the data being used is a collection of time series data. It uses the Dynamic Bayesian Network framework equivalent of a Hidden Markov Model to support activity recognition.

Using these components, a CAPpella allows the user to demonstrate interesting behaviours a number of times and learns from these demonstrations. The user performs a demonstration of a situation and associated action(s) and annotates the captured events, helping a CAPpella to learn. When a CAPpella recognizes the demonstrated situation, it performs the demonstrated actions. a CAPpella supports the automation of actions of only one user.

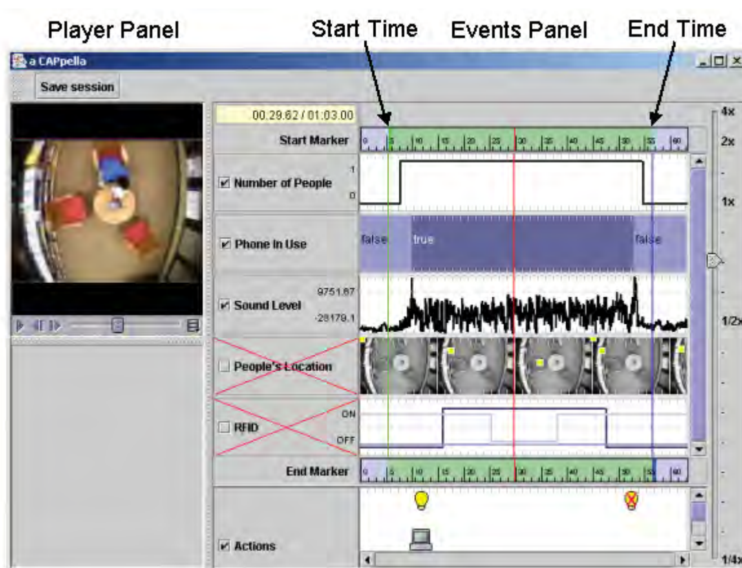

Figure 3.6: a CAPpella user interface

The problem of this approach is that allowing the user to select the relevant events from the recording and using them to train can be tedious for users and has to be repeated a number of times (over a period of days or weeks) before the system learns. In addition, users cannot physically do all they may want to be automated: the programming by demonstration technique is not always valid. Furthermore, although the end-users show the system the behaviour that must be automated, the control is in the system; if the learned behaviour is wrong, end-users only can

A Cappella		
Modelling	Type of model	X
	Expressivity	X
	User Participation	X
	Supported by tools	X
Implementation	Technique / Method	Demonstrate the behaviour to be automated to the system, with prediction help
	Developers vs end-users	End-users
	Supported by tools	Yes
Automation	Capabilities	Sequence of actions triggered when a situation is satisfied
	Dynamic representation of the automations	Hidden Markov Model
	Scalability	-
	Control (users vs system)	System
Runtime Evolution	Level of support	X
	Developers vs end-users	X
	Level of abstraction	X
Domain		Independent
Number of Users	Only one vs multiple users	Only one
Limitations		- Only supports the automation of behaviour that can be demonstrated by users - It may be tedious for end-users to specify the system - The control is in the system - Not support for the evolution of the rules over time

Table 3.8: Table that summarizes the most important characteristics of a CAPpella.

show the behaviour again, they cannot directly modify the learned behaviour.

Table 3.8 summarizes the relevant information of this approach according to the presented layout.

Capture and Access Magnetic Poetry (CAMP). CAMP (Truong *et al.*, 2004) is an end-user programming environment that allows users to create context-aware applications for home. CAMP is

based on a magnetic poetry metaphor that allows users to create a "poem", i.e., a sentence codifying a control statement. This sentence focuses on tasks or goals as users choose using a subset of natural language.

CAMP provides a GUI (see Figure 3.7) to assist users in forming the control statements. The words or pieces are shown to the user in the upper frame of the interface classified in four different categories (who, what, where, when) using different colours. Users can select these words by clicking on them and dragging them down to the poem authoring area on the interface. They can move and re-order words as desired.

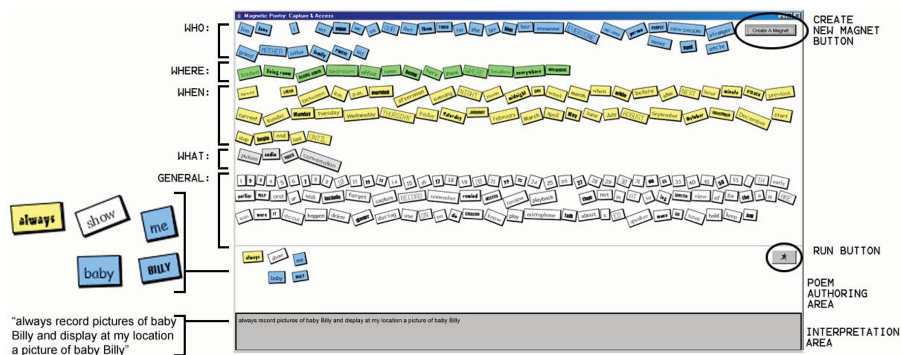

Figure 3.7: CAMP user interface

CAMP supports a limited number of artefacts or data types (i.e., still-pictures, audio, and video) and actions (i.e., capture, access, and delete). For example, a specified sentence could be: "always show me where baby Billy is". In addition, the authors recognize that the CAMP interface cannot scale to display and parse an exhaustive vocabulary.

After creating a poem for the desired application, the user must click the "run" button, which prompts the interface to read the poem and generate a text-based parsing that is displayed in the bottom frame of interface as feedback to the user.

CAMP		
Modelling	Type of model	X
	Expressivity	X
	User Participation	X
	Supported by tools	X
Implementation	Technique / Method	Magnetic poetry metaphor. Specifying the rules as sentences, the approach automatically generates the application specifications and provides the INCA infrastructure for implementing these specifications
	Developers vs end-users	End-users
	Supported by tools	Yes
Automation	Capabilities	Supports the start and stop of capture and access when two specific context conditions occur: time and presence or absence of a person at a location
	Dynamic representation of the automations	X
	Scalability	Low
	Control (end-users vs system)	End-users
Runtime Evolution	Level of support	X
	Developers vs end-users	X
	Level of abstraction	X
Domain		Capture applications
Number of Users		X
Limitations		- Limited for programming applications of capture of audio and video - Difficult scalability to display and parse an exhaustive vocabulary - Not support for the evolution of the rules over time

Table 3.9: Table that summarizes the most important characteristics of CAMP.

Once the end-user has composed a sentence, the system automatically translates it into instructions and parameters for devices, using a custom dictionary to reword and restructure the

user's terms into a format that can be parsed. This translation is displayed in the bottom frame of the interface as feedback to the user. This allows the user to debug the sentences if needed.

The CAMP interface is built on top of the INCA infrastructure (Truong & Abowd, 2004). It abstracts the lower level details involved in the development of capture and access applications, and provides customizable building blocks that support interfaces for capturing and accessing information, components for storing information, a way to integrate relevant streams of information, and the removal of unwanted data.

Thus, CAMP provides a great balance between simplicity and expressivity; however, its domain is restricted to programming capture applications such as “Record picture in Billy's bedroom at night”.

Table 3.9 summarizes the relevant information of this approach according to the presented layout.

Alfred. Alfred (Gajos *et al.*, 2002) is an end-user programming interface that gives the user the ability to program the system to her particular needs and preferences. This interface is built within the Intelligent Room Project (Nicholas Hanssens & Horton, 2002), and using the Rascal (Gajos, 2001) and ReBa (Kulkarni, 2002) systems, which are responsible for the adaptive and reactive components.

Rascal is a framework for goal-directed self-adaptivity. It works with goals, plans for achieving those goals, and the resources needed to implement the plans. Since each goal may be satisfied by multiple plans, the system can select the most appropriate plan according to the context and user's preferences.

ReBa responds to events from the environment's perceptual systems. For example, it might turn on lights if someone entered the room. Reactions are handled by posting a new goal for Rascal to achieve; the goal is then satisfied by finding an appropriate plan. For example, when a user enters his office in the morning,

the system might react by posting the goal of illuminating the room. One plan for illuminating the room might turn on the lights, while another might open the drapes.

By using these components, Alfred allows an end-user to "program" macros by using verbal commands or other natural forms of interaction. A macro is a simple sequence of tasks. Thus, Alfred is a multi-modal macro recorder. To record a macro, the user must tell Alfred the name of a new goal, demonstrate one or more plans for achieving that goal, and finally tell the system the conditions under which it would prefer one plan to another. The following dialogue shows an example of how a macro is recorded:

User: I want to record a new macro.

Computer: Beginning to record a macro. Say "stop recording" when you are done.

User: Turn on the main lights. Open the drapes. Turn on my desk lamp. Say "good morning". Stop recording.

Computer: What phrase would you like to associate with this macro?

User: Good morning, computer.

Computer: Any other phrase?

User: No, I am done.

Computer: Macro added!

Every time the user names the macro, the system executes the programmed tasks for that macro. Also, users can add some hardware trigger to the macro (i.e., when I press this button run the "Good morning, computer" macro). The recorded macros are simple task sequences lacking explicit conditionals. However macros can call other macros, giving users the capability to create abstractions.

Alfred has been successfully evaluated in a conference room and in offices, always with only one user.

Alfred		
Modelling	Type of model	X
	Expressivity	X
	User Participation	X
	Supported by tools	X
Implementation	Technique / Method	Recording the tasks to be automated using verbal commands or other natural forms of interaction. The system automates the tasks as recorded.
	Developers vs end-users	End-users
	Supported by tools	Yes
Automation	Capabilities	Sequence of actions reactive when an event happens. Possibility to use abstraction hierarchy.
	Dynamic representation of the automations	Macros
	Scalability	-
	Control (users vs system)	End-users
Runtime Evolution	Level of support	X
	Developers vs end-users	X
	Level of abstraction	X
Domain		Independent. Demonstrated in offices and in a conference room
Number of Users	Only one vs multiple users	Only one
Limitations		- The user has to know the vocabulary to record the tasks - Lack of conditionals and context-adaptivity - Not support for the evolution of the rules over time

Table 3.10: Table that summarizes the most important characteristics of Alfred.

While this system is perfectly suited to enhance direct interaction and applies some valuable ideas for multi-modal interaction, it lacks the potential to design more complex context-aware applications due to mainly the lack of conditionals among tasks

(i.e., Alfred focuses on giving support for executing a sequence of tasks) and the explicit management of context to configure the automated actions.

Table 3.10 summarizes the relevant information of this approach according to the presented layout.

Pervasive Interactive Programming (PiP). Pervasive Interactive Programming (PiP) (Chin *et al.*, 2008) provides a platform aimed at non-technical people to customise the functionality of their digital home to suit their particular needs.

PiP is based on the concept of a MAp. A MAp contains a collection of rules that determine the behaviour of the environment. Rules has two parts: the *Antecedent* (which are the conditions that enable de rule) and *Consequent* (which is the actions that are executed if the conditions are satisfied). It is worth mentioning that PiP assumes that the logical sequence of actions is not important.

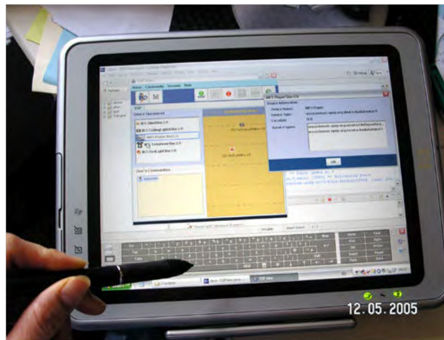

Figure 3.8: PiP Graphical Interface

In order to create a MAp, the user can use any of the following three methods: (1) physically interacting with the devices themselves by demonstrating the functionalities that the MAp should have via simple familiar interaction (e.g., by using a wall switch to turn on a light), (2) using a UI control panels (which are

shown in Figure 3.8) that allows the user to "drag & drop" device representations by engaging them in graphical activities; (3) a combination of the above two methods. To terminate a MAp, the user simply clicks on the "stop" button of the interface.

PiP leverages ontology semantics as the core vocabulary for its information space. When the user finishes the creation of a MAp, PiP generates its corresponding ontology-based rule set and the appropriate events and passes them to the network.

MAps can be visible to the user who created them, either at the time of creation or later, when they can be retrieved, shared, executed, or removed on demand. For instance, to execute a MAp, the user only needs to drag the MAp graphical representation and drop it into a "play" button located at the top of the PiP View.

The communication between PiP, the end-user and the environment is via an eventing mechanism; thus, PiP has an event-based object-oriented asynchronous architecture and leverages $UPnP^{TM}$ technology as its middleware and communication protocol.

This platform has been evaluated by using the iDorm (Hagras *et al.*, 2004) infrastructure obtaining good usability results.

Although this approach provides great control of the system to end-users, it still lacks of important capabilities such as order or relationships among the automated actions or the explicit management of context for configuring these actions.

Table 3.11 summarizes the relevant information of this approach according to the presented layout.

3.4.1 Analysis and Discussion

End-user programming techniques generally provide better user control than the rest of approaches. This improves user acceptance of the system. However, these approaches provide limited capacities to allow end-users to know how the automations must be built. End-users only

PiP		
Modelling	Type of model	X
	Expressivity	X
	User Participation	X
	Supported by tools	X
Implementation	Technique / Method	Programming by example or using a UI control Panels. The rules are automatically generated.
	Developers vs end-users	End-users
	Supported by tools	Yes
Automation	Capabilities	Condition-action rules
	Dynamic representation of the automations	X
	Scalability	X
	Control (end-users vs system)	End-users
Runtime Evolution	Level of support	Create and delete rules
	Developers vs end-users	End-users
	Level of abstraction	Low
Domain		Home
Number of Users		Only one
Limitations		- Low expressivity: not adaptation to context or relationships between tasks. - Not order among the automated tasks - Limited evolution support

Table 3.11: Table that summarizes the most important characteristics of PiP.

can program basic event action rules. For this reason, they are only appropriate for developing simple tasks commonly described in the literature, such as controlling lights or creating doorbells.

In our approach, end-users are always in control of the automations because they participate in their design and their evolution. But, unlike end-user programming techniques, our proposal claims for an approach in which analysts and end-users cooperate for obtaining the

desired automations. The collaboration of end-users is really important to minimize the mismatch between their expectations and the system behaviour, but the figure of analysts is also very important because their knowledge and experience are essential for obtaining a better result. In addition, the cooperation between analysts and end-users allows end-users to better understand how the system and the tools work, favouring they successfully adopt the system and use it.

A further problem that end-user programming techniques present is that most of them deals only with how to train or develop the system initially, not with how to override or modify behaviour later (for example, when unexpected actions arise or user requirements change). This problem is improved in our approach, in which end-users can evolve the automations at runtime to adapt them according to their needs.

3.5 Benefits of our Proposal

In order to automate user behaviour patterns, our proposal claims for the cooperation among analysts and end-users. The collaboration of end-users helps to minimize the mismatch between their expectations and the system behaviour, and analysts apply their knowledge and experience for obtaining a better result. To allow this cooperation, we propose to specify the behaviour patterns to be automated at a high level of abstraction using a context-adaptive task model and a context model. These models are specified by analysts, and are refined and validated with end-users' participation.

This approach brings the following main benefits:

- any behaviour pattern can be automated regardless users performed them before.
- behaviour patterns are analysed before automated, therefore, they can be specified to be automated more efficiently in time and energy concerns.
- users' desires are taken into account in the behaviour pattern specification because users participate in it.

To automate these behaviour patterns, our approach takes into account that user behaviour may change over time and, therefore, their specified behaviour patterns must be evolved to adapt to these changes. To facilitate this evolution at runtime and after system deployment, the models are directly interpreted at runtime to automate the behaviour patterns as specified. This makes the models the unique representation of the automated user behaviour patterns. Thus, if user behaviour changes over time, the system can be easily evolved: the automated behaviour patterns can be adapted by only updating the models. Our approach provides mechanisms for supporting this evolution at runtime and at a high level of abstraction. These mechanisms also facilitate to provide end-user tools that allow the own end-users to adapt their automated behaviour according to their needs.

Thus, we attempt that users are involved in all the process; therefore, the system only automates the tasks that users want to be automated (users are always in control of the system), facilitating user acceptance.

3.6 Discussion and Conclusions

This chapter has presented the state of the art in proposals that deal with intelligently acting on behalf of users according to context. We have revisited and explained the most important approaches by classifying them in three categories: machine-learning approaches, rule-based context-aware approaches, and end-user centred approaches.

As well as analyzing and discussing the limitations that each category presents, we have shown, for each approach, the most important characteristics that are relevant to achieve the challenges confronted in this thesis. As described in Section 1.3, these challenges are the modelling, automation and evolution of behaviour patterns.

Table 3.12 summarizes the level of achievement of these challenges by each one of the approaches described in this chapter. As shown, none of these approaches attempt to confront the automation of user actions at a high level of abstraction, i.e., none of the revisited

	Modelling	Implementation	Automation	Runtime Evolution
NNH	X	Automatic learning	Sequence of actions with limited functions. Control in the system	X
iDorm	X	Automatic learning	Event-action rules. Control in the system	Automatic evolution of the rules. Low level
MavHome	X	Automatic learning	Sequence of actions with temporal relationships. Control in both	Modification of the rules automatically and by end-users. Low level
ParcTab	X	Manual implemented	Condition-action rules. Control in the system	X
CML	Graphical context model	Manual implemented	Context triggered actions. Control in the system	X
García-Herranz	X	Manual implemented or end-user programming	ECA rules with timers. End-users' control	X
a CAPpella	X	End-user programming and automatic learning	Context triggered actions. Control in the system	X
CAMP	X	End-user programming	Capture and Access actions. End-users' control	X
Alfred	X	End-user programming	Event-action rules. End-users' control	X
PiP	X	End-user programming	Condition-action rules. End-users' control	Create and delete rules. Low level

Table 3.12: Table that summarizes the state of the art of the challenges confronted in this thesis.

approaches proposes models to capture the automations at a high level of abstraction. Only the language CML proposes a high level of abstraction model, however, it is focused on specifying the context managed by the system, not the automations.

Regarding the development and automation of user behaviour patterns, most of the approaches provide great facilities for developing and automating sequence of actions. Only a few of the presented

approaches allow temporal relationships and abstraction hierarchies to be established among these actions. However, none of them provide the expressivity and the automation capabilities that our approach achieves, i.e., none support the use of context for establishing context conditions (context relationships among tasks, context preconditions, context parameters to execute a task) to configure how and when the automated actions must be executed. This expressivity allows us to form behaviour patterns that group together the tasks that must be executed to achieve a whole goal (e.g., tasks for waking up the user, tasks that must be carried out when users leave home, etc.), which improves the understanding and management of the automated tasks.

Finally, regarding the evolution of the behaviour patterns that are automated, only two of the presented approaches support runtime evolution. However, these approaches support this evolution at a low level of abstraction. Unlike them, in this thesis we attempt to confront the runtime evolution of the automated behaviour patterns at modelling level, by using the same high level concepts used for creating the designed models.

Thus, in spite of the research efforts that have been done, this chapter shows that there is still work to be done in order to completely solve the challenges confronted in this thesis.

Chapter 4

Overview of the Proposal

Since the advent of Pervasive and Ubiquitous computing, the automation of user routine tasks, also well-known as behaviour patterns, has been a pursued challenge. Its achievement could not only reduce the tasks that users must perform everyday, but also performing them in a more comfortable way for users and optimizing energy and water consumption. In order to deal with this challenge, our approach makes use of the pervasive services provided by any pervasive system.

A pervasive system is developed to provide omnipresent services that serve people in their everyday lives. These services are in charge of interacting with physical devices in order to change the state of the environment and to sense context. Thus, services are considered as the primary elements of the pervasive system architecture.

Instead of considering these services in an isolated manner, we propose to support the automation of user behaviour patterns by coordinating these services in a context-adaptive way. Thus, the automation of a behaviour pattern could be seen as a specific coordination of pervasive services that is performed in the opportune

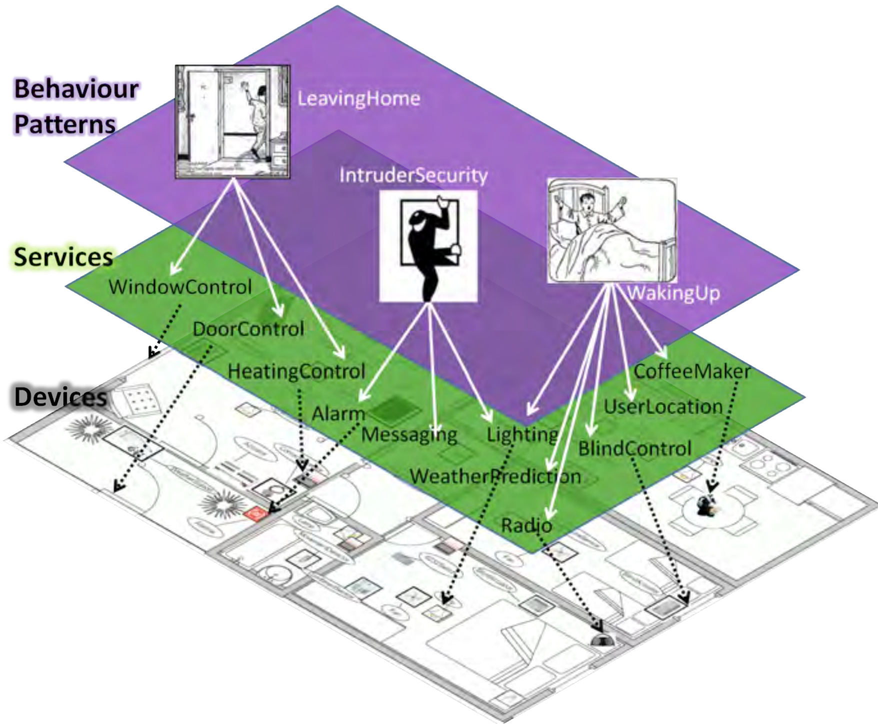

Figure 4.1: Pervasive System Architecture

context. Figure 4.1 represents this notion in a graphical way. Behaviour patterns coordinate services to meet user needs, while these services use the devices installed in the environment to provide their functionality. For instance, the WakingUp behaviour pattern for waking up the user can turn on the radio by using the *Radio* service. Then, if it is a sunny day, the pattern can raise the bedroom blinds using the *BlindControl* service, or can switch on the bedroom light using the *Lighting* service if outside light intensity it is not enough. Afterwards, when the user is in the kitchen, which is known using the *UserLocation* service, the pattern can make breakfast using the *CoffeeMaker* service and inform the user about the weather using the *WeatherPrediction* service.

Although several approaches have confronted the automation of user

behaviour patterns, the provided solutions still present some drawbacks that we attempt to solve in this work. In this chapter, we present an overview of this solution. We firstly introduce it in a nutshell. Then, we describe the process that must be followed to achieve the automation of user behaviour patterns using our solution. We then explain the developed software infrastructure to support this process. Afterwards, we show how our solution has been put into practice and validated throughout several case studies. Finally, we conclude the chapter.

4.1 Introduction

As explained in Chapter 3, although several approaches have dealt with the pursued challenge of automating user behaviour patterns, they still present some drawbacks. The two most important ones are the intrusiveness of the automated tasks and their maintenance and evolution over time.

Regarding intrusiveness, we have to consider that the automation of users' tasks on their behalf is a very delicate matter. The execution of a not desired task may be very intrusive for users, bothering them, interfering in their goals or even being dangerous. To avoid these problems users' desires and demands have to be taken into account in order to automate the tasks that users want in the way they want them.

Regarding maintenance and evolution, we have to consider that users' behaviour may change over time. Therefore, the automation of their behaviour patterns must be adapted to these changes; otherwise, system may become useless, obsolete, and intrusive. To avoid this, the automation of user behaviour patterns has to be done to facilitate their further evolution, and also evolution mechanisms have to be provided.

To solve these problems, we propose a context-aware model-driven approach that achieves the automation of the behaviour patterns that users want to be automated.

In order to avoid intrusiveness, it is essential the collaboration of end-users in the obtaining of the behaviour patterns to be automated.

In this thesis, we deal with this goal by proposing two models of a high level of abstraction. This allows end-users to focus on the main concepts (the abstractions) without being confused by many low-level details (Paternò, 2003).

In order to facilitate the maintenance and evolution of the automated behaviour patterns, we use the MDE principles (see Section Section 2.2). The objective of the proposed models is not only to analyse the behaviour patterns to be automated, but become into the primary means to understand, interact with, and modify the behaviour patterns. To achieve this, the models are designed to be machine-processable and precise-enough to be executable models.

To properly automate the specified behaviour patterns, we design and develop a software infrastructure that performs the behaviour patterns by executing the models at runtime. This software infrastructure could execute the models by following two strategies: code generation or model interpretation. While code generation makes that the system has to be stopped to be generated, packaged and installed again when new changes must be applied, model interpretation makes the evolution much more easier. Using model interpretation, the models are the only representation of the automated behaviour patterns; therefore, to evolve them, only the models must be updated. For these reasons, the proposed software infrastructure directly interprets the models to execute the behaviour patterns. This greatly facilitates the runtime evolution of the behaviour patterns specified to be automated.

In addition, to support the runtime evolution of the behaviour patterns, our approach provides mechanisms and tools to update the automated behaviour patterns according to user needs.

4.2 Process for Automating User Behaviour Patterns

To achieve the automation of user behaviour patterns by using our approach, we propose a model-driven development process that benefits from the whole range of gains brought by the application of MDE

and model interpretation (see Chapter 2.2). For instance, we obtain important benefits as automation of the development, support for the simulation and early requirement validation, reusability, technology independence, ease to perform changes at runtime, etc.

To define the process, we use Software and Systems Process Engineering Meta-Model 2.0 (SPEM 2.0), which is the OMG standard for describing development processes. Next, we first explain briefly the notation of this language and then we describe the proposed process in detail.

4.2.1 SPEM notation

SPEM 2.0 is used to define software and systems development processes and their components. Using SPEM, the software process is defined by means of a set of activities. Each of these activities is performed by one or more process roles and can be divided into steps by depicting a composite relationship from the activity to the step. After the performance of each activity or step, one or more work products can be obtained (output work products of the activity). An activity or step can also require some work products in order to be performed. These work products can be either output work products of other activities or work products already performed in other processes (input work products of the activity). Also, an activity can update some work product in its development. The notation proposed by SPEM in order to represent all these software process elements is presented in Figure 4.2.

Additionally, SPEM proposes the use of UML 2.0 activity diagrams[1] in order to define sequences of activities as well as their input and output work products. In this case, nodes in activity diagrams represent activities or work products. Arcs in activity diagrams represent: (1) a sequence of activities (depicted by solid arrows) if both the source and the target of the arc are activities or (2) output or input work products of an activity (depicted by dashed arrows) if the target or the source of the arc is a work product. Solid arrows are also used to indicate

[1] http://www.uml.org/

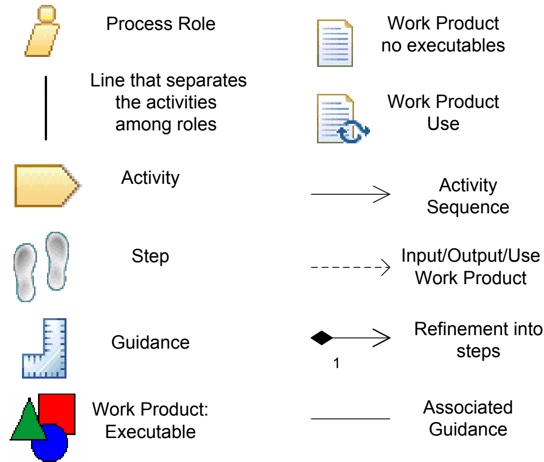

Figure 4.2: SPEM notation

the initial activity within a software process. To do this, these arrows connect the initial activity with the process role that performs it. Figure 4.2 shows the notation of these arcs.

SPEM also allows us to associate activities with guidance elements that help to perform them. As shown in Figure 4.2, this association is represented by a solid line, which is depicted between the activity and the guidance.

4.2.2 The Process Activities

The proposed process is composed of a sequence of activities to be followed. These activities are driven by the challenges confronted in this thesis, deriving each challenge in one activity of the development process: modelling, automation and evolution. These activities must be performed after the first one, which is the requirement elicitation for identifying the behaviour patterns that users want to be automated. Thus, the process, which is represented using SPEM in Figure 4.3, is divided in the following activities:

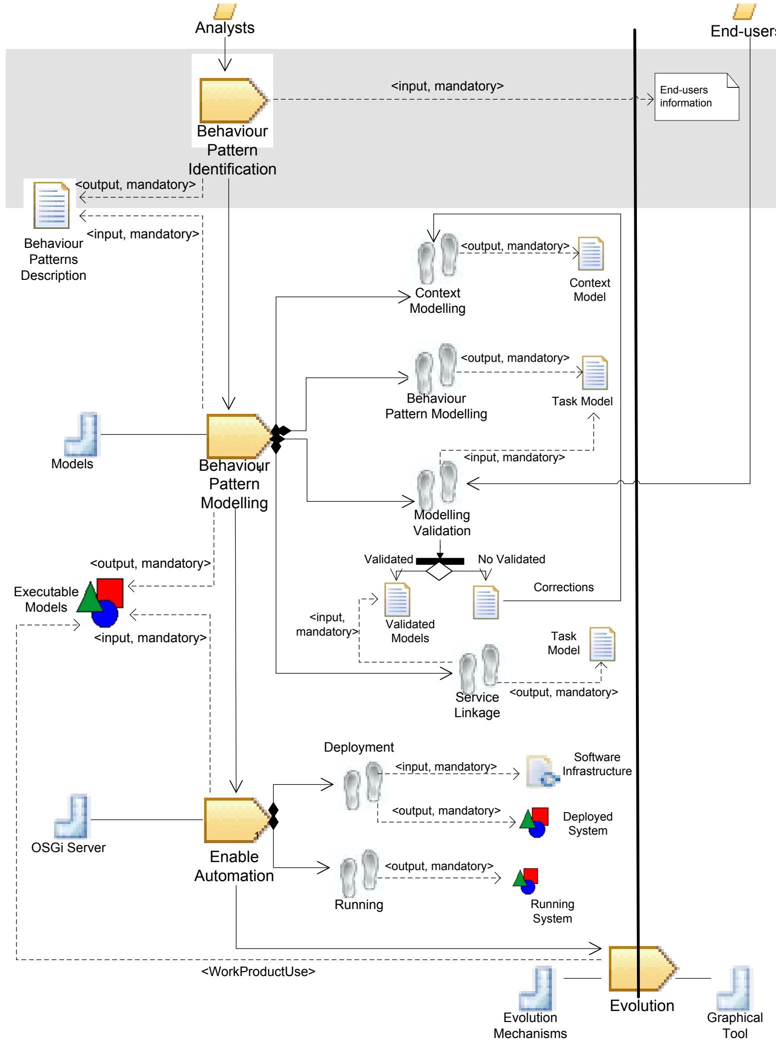

Figure 4.3: SPEM Process for achieving the automation of user behaviour patterns

Behaviour Pattern Identification. Analysts interview users to determine the behaviour patterns that they want to be automated. This activity is shown in the figure with a grey background because it falls out of the scope of this work. Any requirement elicitation process can be used to identify them. Particularly, we have used scenarios to capture the automation requirements. Scenarios are a well-known technique often used during the initial informal analysis phase. They provide informal descriptions of a specific use in a specific context of an application. A careful identification of a set of meaningful scenarios allows analysts to obtain a description of most of the activities that should be considered.

Behaviour Pattern Modelling. This activity consists of modelling the behaviour patterns that must be automated by the system. A behaviour pattern is a set of tasks that are habitually performed in the similar contexts. For this reason, context information must be also specified to be able to carry out the behaviour patterns in a non-intrusive way. Thus, to specify the identified behaviour patterns, the following steps must be followed:

- Context modelling: analysts specify the context properties on which the behaviour patterns depend.
- Behaviour pattern modelling: analysts specify the behaviour patterns to be automated according to the context previously specified. Each behaviour pattern is specified as a hierarchy of tasks adaptive to context.
- Modelling validation: the behaviour pattern modelling is validated with the end-users to ensure that the tasks that are going to be automated are those tasks that users want and these tasks are going to be automated in the way users want. Thus, following an iterative process, the task modelling (and if needed the context modelling), must be refined with end-users' participation until they agree with the specified behaviour patterns. It is important to note that, in this way, the modelling is complemented by both the analysts'

knowledge, which contributes to improving the performance of the identified behaviour patterns; and users' knowledge, which contributes to taking into account their demands and desires. After validating the task modelling with user participation, analysts also validate that the models are correctly formed and inconsistencies are not found in them.

- Service linkage: once the modelling has been validated, the analysts link each pattern task to be executed, with a pervasive service that can carry it out. This linkage is made in the task model by indicating the name of the corresponding service. To be able to be applied, our approach needs pervasive services in charge of controlling the devices of the environment (e.g., switching lights on, activating the security alarm, etc.) as well as of sensing context information (e.g., detection of presence, measurement of temperature, etc). The implementation of these services is out of the scope of this thesis. In particular, we have used a modelling approach named PervML (Muñoz *et al.*, April 2006; Serral *et al.*, 2010) for developing them. More detail about the development of the pervasive services will be explained in Chapter 6.

Since we use model interpretation to automate the specified behaviour patterns, this step finishes the development of the automations. This also allows that the specified behaviour patterns can be validated by using prototypes. This would require that the automation of the specified patterns is done in a simulation mode. This mode should allow us to cause context changes and to easily observe the execution of the behaviour patterns according to context.

Automation of the specified behaviour patterns. To enable the automation of the specified behaviour patterns in the opportune context, the following two steps must be performed:

- Deployment of the system in the target platform. To

deploy the system, analysts install each component of the software infrastructure in an OSGi server. We use an OSGi server (OSGI, 2011) because it provides numerous benefits and facilities to make dynamic updates, to easily reuse components, or to deploy the system. See all the benefits of OSGi in Section 2.4. In addition, the files in which the behaviour patterns are specified must be saved in the folder where OSGi is installed.

- Running the system. To run the system, analysts start the components installed in OSGi. From this moment, the context is continuously monitored by a context monitor that reflects the context changes in the context modelling. This monitor also notifies an automation engine about these changes. In this way, the engine can check whether some behaviour pattern must be carried out in the new context. If some behaviour pattern must be carried out, the engine is in charge of executing it by interpreting the models at runtime. It is important to note that, since the models are not translated to code but they are directly interpreted by the automation engine, the models are the only representation of the behaviour patterns to be automated. This facilitates their understanding and maintenance.

Evolution of the behaviour patterns if needed. User behaviour may change over time and the behaviour patterns that are automated may become obsolete or useless. If this happens, the automated behaviour patterns can be adapted according to user needs. To allow this evolution, evolution mechanisms and a graphical tool are provided.

4.3 Software Infrastructure

To support the automation of user behaviour patterns by following the process explained above, we design and develop a set of software components. Specifically, these components support the application of

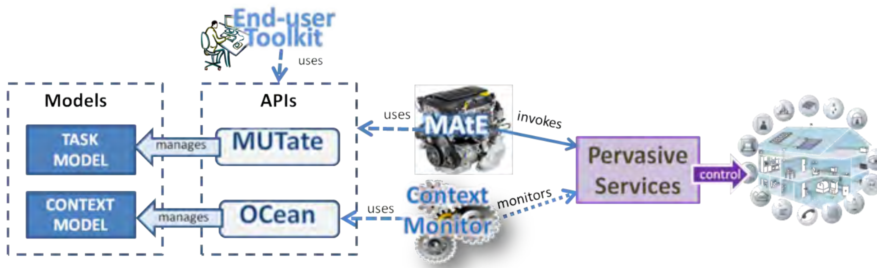

Figure 4.4: Software infrastructure

the three steps of the process derived from the three goals pursued in this thesis: modelling, automation and evolution of user behaviour patterns.

Figure 4.4 presents an overview of these components. This figure also shows the relation of these components with the pervasive services in charge of controlling the devices of the environment. For each one of the developed software components, we provide the following information: name of the component, a brief description, steps of the process that supports and the chapters where the component will be explained in detail.

Models: To specify the behaviour patterns to be automated, we propose two models: one for specifying the context needed for performing the behaviour patterns unobtrusively and other for specifying each behaviour pattern as a composition of tasks adaptive to context.

To specify the context information needed, we propose an ontology-based context model. This model is based on an ontology because it is one of the best models to specify context (Baldauf *et al.*, 2007; Chen *et al.*, 2004; Ye *et al.*, 2007). Specifically, the context model proposed in this thesis describes the context needed for properly automating the identified behaviour patterns. For facilitating the specification of the context model, the Protégé

tool is used. Protégé is a free open source ontology editor and knowledge-base framework.

To specify the behaviour pattern according to the context specified in the context model, we propose a context-adaptive task model because: 1) it can provide enough expressivity and precision, which is needed to be automatically executed; and 2) it can be understood by end-users (Johnson, 1999; Lauesen, 2003), which is needed for validating the model with their participation. The task model specifies each behaviour pattern by describing: the context situation in which the pattern has to be executed, the tasks to be executed for each one of the identified patterns in a hierarchical way (from more general to more specific), and the temporal relationships that must be accomplished for the execution of these tasks. In the task model, the behaviour pattern tasks and their relationships are specified using the context information of the context model, in such a way that the task execution automatically adapts according to context. For facilitating the specification of the task model, we develop an Eclipse(Eclipse, 2011) modelling tool that allows the task model to be graphically specified.

According to the classification described in 2.2.1, these models are development models, because they are used to develop the automations of systems; they are also executable models because they are fully expressive to be automatically executed; and they are also runtime models, because they are directly interpreted at runtime to perform the specified automations as specified in the models.

Step/s that supports: the proposed models support the modelling step.

Chapters where the component will be detailed: the context and task models will be described in detail in Chapter 5.

Application Programming Interfaces (APIs): Two APIs are provided in order to manage the models at runtime: (1) MUTate

(Model-based User Task management mechanisms), which provides constructors to manage the task model; and (2) OCean (Ontology-based Context model management mechanisms), which provides constructors to manage the context model. To facilitate the use of these API's, the constructors that they provide use the same high-level concepts used for creating the models.

Step/s that supports: The APIs are used to support the automation and evolution steps.

Chapters where the component will be detailed: in Chapter 6 these APIs are explained. In Chapter 7 it is explained how these APIs are used for evolving the behaviour patterns.

Context monitor: The context monitor is in charge of capturing and processing context changes and then updating the context model accordingly. Note that these changes are physically detected by sensors, which are controlled by pervasive services. Thus, in order to capture context changes, the monitor is continuously monitoring the execution of the pervasive services. When a change in context is detected, the context monitor reflects the change in the context model by using OCean. Next, the context monitor is in charge of informing the automation engine (which is introduced below) about the context that has been updated.

Step/s that supports: the context monitor is used to support the automation step.

Chapters where the component will be detailed: the context monitor will be described in detail in Chapter 6.

Model-based user task Automation Engine (MAtE): MAtE is in charge of executing the corresponding behaviour patterns in the appropriate context as specified in the models. When MAtE receives the context change notification sent by the context monitor, MAtE checks whether there is any behaviour pattern that has to be executed. To do this, it interprets the context model and the task model by using the provided MUTate and

OCean APIs. If a pattern has to be carried out, MAtE executes the corresponding pervasive services as specified in the task model.

Step/s that supports: MAtE is used to support the automation step.

Chapters where the component will be detailed: MAtE will be explained in detail in Chapter 6.

Evolution tool: A high-level tool is provided for allowing the evolution of the automated behaviour patterns after system deployment and without the need to stop or redeploy the system. Since MAtE directly interprets the models at runtime to automate the behaviour patterns, as soon as the models are changed, the changes are applied. Thus, this tool provides users with a set of intuitive interfaces that allow them to change the models at runtime. The tool uses MUTate and OCean to update the models at runtime.

Step/s that supports: the evolution tool is provided to support the evolution step.

Chapters where the component will be detailed: the evolution tool will be explained in detail in Chapter 7.

4.4 Validation

The presented work has been validated from three different perspectives according to the confronted challenges:

Modelling of the behaviour patterns. The task model and the context model must provide enough expressivity to specify the behaviour patterns that users want to be automated. Also, this specification must be understandable enough for the end-users so that the models become artefacts for discussion between analysts and users. Experimentation results show that, although some specific aspects are not very intuitive to be described, all the

identified behaviour patterns can be specified using the provided models. Regarding the comprehension of the task model, all the users could understand and reason about the routines specified in the model.

Automation Infrastructure. The provided infrastructure must automate the behaviour patterns in the opportune context as specified in the models. Experimentation revealed that the automation of user tasks is properly performed using our software infrastructure. Also, model interpretation must be subject to the same efficiency requirements as the rest of the system because this impacts overall system performance. Therefore, we analysed the scalability of our approach by studying the temporal cost of the operations that access models. Experimentation results showed that the user routine tasks are automated without drastically affecting the system response.

Evolution of the Automated Patterns. The mechanisms and the tool provided for evolving the automated behaviour patterns must allow them to be easily evolved according to users needs, and at runtime. As well as testing the evolution mechanisms, we perform an experiment with end-users to evolve the automated behaviour patterns. This experiment revealed that most of the users were capable of performing the required evolutions efficiently.

To evaluate the above concerns, we have performed a case study based evaluation. To perform it, we have developed several smart home case studies and a nursing home case study (see Appendix B) following the guidelines for case study research by Runeson and Höst (Runeson & Höst, 2009).

From this evaluation, we conclude that the approach achieved satisfactory results regarding the automation of user behaviour patterns. Nevertheless, we detected that some aspects of the task model were still a little difficult to understand by some users. In addition, although all the users achieved to evolve the behaviour patterns using the provided evolution tool, some of them took it too much effort to understand how

some aspects of the tool worked. These aspects need to be improved to provide more facilities.

4.5 Conclusions

The automation of user routine tasks is a very desirable challenge because it can considerably improve users' quality of life. However, this challenge is also very difficult to properly achieve because the automation of user tasks may become intrusive if it is not performed exactly in the way users want it. In this thesis, we propose an approach that attempts to take into account users desires and demands overall the development process. This process uses model interpretation achieving that: 1) the user routines are specified at a high level of abstraction and are obtained by simply specifying them in models; 2) the evolution of the routines over time can be performed at a high level of abstraction and at runtime. Finally, we have evaluated our approach by applying it to several case studies, obtaining valuable validation information of the approach.

Chapter 5

Modelling User Behaviour Patterns

Modelling techniques focus on abstract models rather than computer programs. Abstract models allow a system to be designed by using concepts that are much less bound to the underlying implementation technology and are much closer to the problem domain. This makes the models easier to specify, understand, and maintain than computer programs. In addition, the use of abstract models facilitate the participation of end-users in the early stages of the development process because models allow end-users to focus on the main concepts (the abstractions) without being confused by many low-level details (Paternò, 2003). Furthermore, when models are machine-processable and precise-enough, they can be used as executable models to automate the production of a software system (Pastor & Molina, 2007).

These benefits are the main reasons for selecting a modelling technique to achieve the automation of user behaviour patterns. A behaviour pattern is a set of tasks that are habitually performed when similar contexts arise (Neal & Wood, 2007). For instance, a behaviour

pattern could be the set of tasks that we usually perform when leaving home: switching off the lights and air conditioning, closing windows and locking doors; another example could be the set of tasks that we usually perform in the morning: at 7:55 in working days, the heating of the bathroom is switched on, five minutes later the alarm clock goes off, and, when we are in the kitchen, the breakfast is prepared. They are two examples of behaviour patterns.

As the definition states and the examples show, a behaviour pattern is made up of two essential parts: the set of tasks that compose the behaviour pattern (i.e., the alarm clock goes off, the heating is switched on, etc.); and the context on which the behaviour pattern, including their tasks, are performed (i.e., time, users' presence, environment location). Note that knowing context is essential for automating user tasks unobtrusively, i.e., without bothering users.

Thus, as well as specifying the behaviour patterns, the context information needed for carrying out them have to be also specified. Therefore, to specify the behaviour patterns to be automated, we propose two models: a context model and a context-adaptive task model. The context model captures the context information that must be taken into account for automating the behaviour patterns under the opportune conditions. The task model specifies the behaviour patterns by decomposing them into a hierarchy of tasks. In addition, in the task model it is specified the context conditions where the behaviour patterns and their tasks must be carried out. These context conditions are specified by using the context information previously described in the context model.

Section 5.1 describes the proposed context model. Section 5.2 explains the proposed task model. Finally, Section 5.3 presents the conclusions of the chapter.

5.1 Modelling Context

A suitable model for handling, sharing and storing context is essential for automating user behaviour patterns in a non-intrusive way. In

this section, we first study the concept of *context* to establish the information that our context model must capture. Next, we revisit the most important models that have been proposed to model context in order to select a suitable model. Finally, we explain the context model proposed in this thesis to represent the context needed for correctly automating user behaviour patterns.

5.1.1 The Context Concept

The *context* concept is widely used in Computer Science with different meanings according to the working area. For instance:

- In Artificial Intelligence (Lieberman & Selker, 2000), it is defined as everything that affects the computation except from the explicit input and output of the application. According to this definition, what is considered explicit and implicit has to be precisely determined in the system. All what is considered implicit constitute context. In this way, the context of a system changes depending on the initial consideration of explicit and implicit elements.

- In Natural Language Processing (Lenat, 1998), it is understood as all the knowledge that surrounds a specific statement or assertion. In this area, it is important to avoid the danger of taking things "out of context". Assertions true in one context might be false in another. Thus, context in this area is related to the meaning of one sentence with regards to the meaning of other sentences.

- In Operating Systems (Stalling, 2000), it is defined as the minimal set of data used by an operating system task. This data needs to be saved in order to allow the task to be interrupted at a given date, and to be resumed at the point it was interrupted.

In Pervasive Computing, the definition of *Context* is different from the ones presented above. Some of the most important definitions in Pervasive Computing are the following:

- In (Schilit *et al.*, 1994), Context is characterized by the location of use, the collection of nearby people, hosts, and accessible devices. Thus, context-aware systems are those that are able to adapt themselves to these aspects.

- In (Ryan *et al.*, 1998), Context-awareness is described as the ability of the computer to sense and act upon information about its environment, such as location, time, temperature, or user identity. Thus, the concept of Context is mainly centred on the characteristics of user location.

- In (Dey, 2001), the most used definition of Context in AmI systems is presented. Dey defines Context as any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and the application themselves.

- In (Mitchell, 2002), two classes of Context are identified, namely personal and environmental context. Examples of environmental context include: the time of the day, the opening times of attractions and the current weather forecast. Personal context refers to user profiles in which information such as user's interest, attitudes, or beliefs are considered.

- In (Crowley *et al.*, 2002), a distinction between user's context and system's context is done. User's context provides means to determine what to observe and how to interpret the observations. System's context provides means to compose the federation of components that observe the user's context.

- In (Chen *et al.*, 2004), Context is defined as the following information: people, software agents that the system contains, beliefs, desires and intentions of these software agents, actions, policies, time, space, and events.

- In (Bardram, 2005), Context refers to the physical and social situation in which computational devices are embedded.

Although there is not a generally accepted definition for the *Context* term in Pervasive Computing, all of the studied definitions share some ideas about the information that must be considered as Context. Thus, the core context information identified taking into account our automation purposes consists of:

- User information: personal data, preferences, skills, location, mobility, etc.

- Environment information: space information (areas of the environment where the system is running and spatial relations between these areas), environment properties (properties of the environment such as temperature, light intensity, etc.), etc.

- System information: information about the system, such as the services that it provides, the devices of the system, their computational resources, etc.

- Privacy and security policies: information that indicates what actions each user can execute and what context information each user can see and modify.

- Temporal information: date and time, holiday, working day, etc.

- Events' information: information about the events that happen in the system, such as user actions and context changes.

5.1.2 Context Modelling in Pervasive Systems

Different context models have been proposed until now to capture context in Pervasive Computing. Some of the most important examples are: object oriented models such as the proposed by the projects CORTEX (Biegel & Cahill, 2004) and Hydrogen (Hofer *et al.*, 2002); key-value models such as the used by Dey in the Context Toolkit (Dey, 2001); graphical models such as ContextUML (Sheng & Benatallah, 2005), CML (Henricksen & Indulska, 2004) and the proposed in (Ayed *et al.*, 2007); etc.

However, several studies (Baldauf *et al.*, 2007; Chen *et al.*, 2004; Ye *et al.*, 2007) state that the use of ontologies to model context is one of the best choices. They state that this model guarantees a high degree of expressiveness, formality and semantic richness. Ontologies also exhibit prominent advantages for reasoning and reusing context as well as facilitating the integration of pervasive environments. Some relevant examples of ontology-based approaches are SOUPA (Chen *et al.*, 2004), COMANTO (Preuveneers *et al.*, 2004), SOCAM (Gu *et al.*, 2005), and COIVAS (Hervás *et al.*, 2010). A complete background of most of the ontologies proposed in Pervasive Computing can be found in (Ye *et al.*, 2007). None of the studied context ontologies cover adequately all the context information identified in the previous subsection; however, the SOUPA ontology is of special interest for this work.

SOUPA is a proposal for an Standard Ontology for Ubiquitous and Pervasive Applications that defines core concepts by adopting the following different consensus ontologies: FOAF, which captures personal information and social connections to other people; DAML-Time & the Entry Sub-ontology of Time, which represent Time and facilitate the reasoning about the temporal orders of different events; OpenCyc Spatial Ontologies & RCC, which allow space to be specified using geo-spatial coordinates or symbolic representation; Rei Policy Ontology, which specifies high-level rules for granting and revoking the access rights to and from different services.

According to the context-modelling background published in (Ye *et al.*, 2007), SOUPA is the most consistent set of ontologies, since it imports most of its concepts from external and consensual domain ontologies. For this reason, and for facilitating information sharing, we define a context ontology that adequately covers the context information identified in the previous subsection by extending the SOUPA ontology.

5.1.3 An Ontology-based Context Model

The context model that we propose is based on a context ontology that adopts, as far as possible, suitable concepts of the SOUPA ontology, extending it in order to cover all the context information identified in

Section 5.1.1. Thus, to explain the model, we first define the context ontology by identifying the domain concepts and their relationshiops. Next, we explain how we describe the context model based on this ontology.

The context ontology

An ontology is a formal representation of a set of concepts within a domain and the relationships between those concepts. To build the context ontology, we follow a top-down approach, starting from the most coarse-grained concepts and dividing them up into finer-grained concepts. The coarse-grained concepts that we identify are: Environment, System, Person, Policy, Time, and Event. Dividing them into finer-grained concepts, we obtain the classes of the class diagram shown in Figure 5.1.

To describe the **environment**, we reuse the *OpenCyc Spatial* and *RCC* SOUPA ontologies, which allow space to be specified using geo-spatial coordinates or symbolic geographical representation. They propose classes such as *GeographicalSpace* that inherits from *SpatialThing*, whose is related to *LocationCoordinates* class. However, we think that a symbolic representation more intuitive for users is also needed. Thus, we propose the term *Location* to describe the different areas that compose the environment (i.e., Kitchen, Corridor, etc.). A location is characterized by a name and by its relationships with the other locations of the environment. These relationships are *subsumes, adjacency* and *mobility.* The *subsumes* relationship indicates that a location contains other locations (e.g., the location First Floor subsumes the locations Kitchen, Hall and Living Room). The *adjacency* relationship indicates that two locations are physically together (e.g., the Parent Bedroom and the Children Bedroom are adjacent). The *mobility* relationship indicates that two locations are adjacent and there is a way for people to go from one location to the other (e.g., the Hall and the Living Room are adjacent and the Hall has a door to go to the Living Room). In addition, we propose the term *EnvironmentProperty* to describe the properties (e.g., lighting intensity, presence detection,

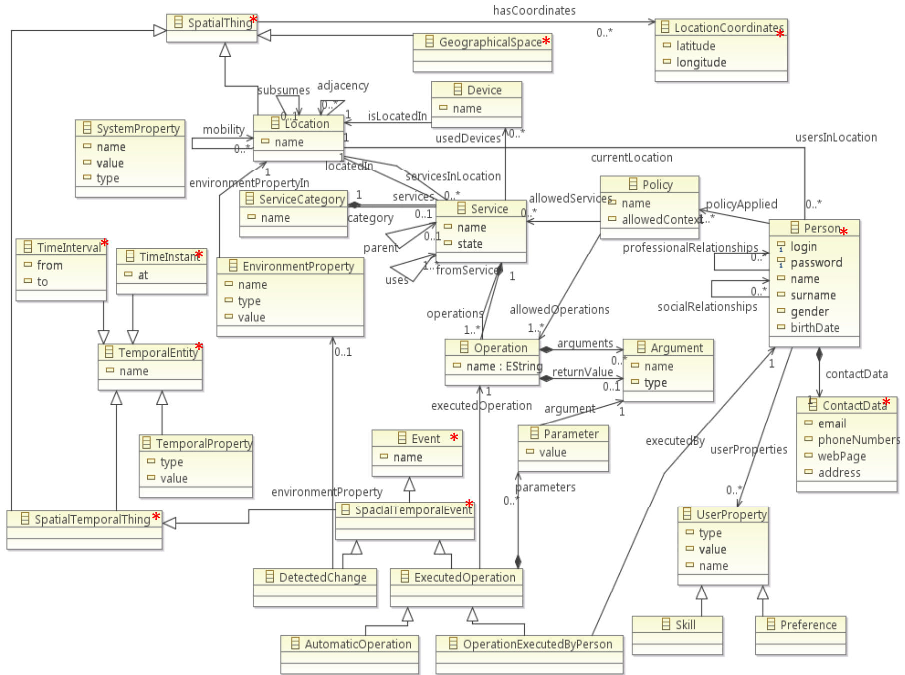

Figure 5.1: Context ontology classes in ecore format

noise level, etc.) of a certain location. Each environment property is characterized by a *name*, a *value*, its data *type*, and the location where is placed.

To describe the **system**, we propose the terms *Service*, *Service-Category*, *Operation*, *Argument*, *Device* and *SystemProperty*. The term *Service* represents the services (e.g., Lighting, Multimedia Player, Alarm, etc.) that the system provides. A service is characterized by the properties *name* and *state* and belongs to a service category (e.g., illumination, multimedia, security, etc). Each service category is characterized by a *name* and can belong to another category by using the *parent* relationship. This aspect allows us to define hierarchies of categories. Services can be related to each other by means of two relationships: *parent* and *uses*. On the one hand, the *parent* relationship

allows us to indicate hierarchies of services by defining parent and child services (e.g., we can define the Lighting service as the parent of the Gradual Lighting service). This relationship is used with inheritance purpose in order to indicate that a service presents all the characteristics of another service plus some additional ones. On the other hand, the *uses* relationship indicates that a service needs to use other service/s in order to provide its functionality (e.g., the HomeLighting service needs to use the lighting services of all the locations contained in the Home location). The behaviour of each service is characterized by its *operations* which are described with a *name* (e.g., the operations of a service Video Player may be play, stop, forward, review, etc), the set of arguments that receives (e.g., the play operation may need the title of the movie) and the argument that returns. In addition, a service is also related to the *devices* used (lamp, dimmer, presence detector, etc.) to carry out its operations. A device is characterized by a *name*. Each service and each device are also related to the location where are provided or installed. The term *SystemProperty* is used for defining specific properties of the system, such as its computational resources, networks, etc.

To describe the **users of the system**, we reuse the *FOAF SOUPA* ontology, which propose the term *Person*. This term is described by a set of properties that include profile information (e.g., *name, gender, birth date*, etc.), contact information (e.g., *email, mailing address, phone numbers*, etc.), and *social* and *professional* relationships (e.g., people that a person knows, relatives, etc.). To properly describe the users, we add the *UserProperty* class, to represent the properties of users, such as its preferences (e.g., preferred music, preferred language, etc), or the skills and disabilities that a person has and may affect to his/her interaction with the system (e.g., computer knowledge, deafness, diseases, etc.). With regard to the location in which a person is, we propose also the *currentLocation* relationship, which relates each person to the location where it is in the current moment.

A *person* is also associated to **policies**. To define the *Policy* term, SOUPA uses the Rei Policy Ontology. It specifies high-level but complex rules for granting and revoking the access rights to and from different

actions (concept similar to the operation concept that our approach provides, but focused only on agents). Since our ontology allows us to describe the services of the system and the operations that they provide, we describe policies in an easier way as a set of operations and/or services (which group a set of operations) that are permitted for a person. In addition, the policy also describes the context information that a person can see and/or modify. In our ontology, the permitted context is defined by indicating the coarse-grained terms proposed in this ontology; e.g., the environment term indicates that all the information about the environment is permitted in the policy. Thus, each policy restricts the operations that a person can use (e.g., we can create a policy for children that does not allow them to activate the security service or the heating service) and the context that s/he can see or modify. A policy is described by a *name*, the set of people to who the policy is applied (*appliedFor*), the set of *operations/services* that the policy allows, and the list of context information that the person can see and modify.

To describe **temporal aspects**, we reuse the DAML-Time ontology and the Entry Sub-ontology of Time that SOUPA provides. These ontologies provide us with the term *TemporalEntity*, which is refined into *TimeInstant* and *TimeInterval*. The *TimeInstant* term is defined by using the *at* property that stores the value of time; while the *TimeInterval* term is defined by using the *from* and *to* properties that relate the time interval to the two corresponding time instants. In addition, these SOUPA ontologies provide useful temporal relationships to compare and order to different temporal entities, for instance: *after, before, sameTimeAs, startsLaterThan, startsSoonerThan, startsSameTimeAs, endsLaterThan, endsSoonerThan, endsSameTimeAs.* For avoiding overloading the model, we do not show these relationships in Figure 5.1. To these classes, we added the *TemporalProperty* class as another refinement of the TemporalEntity class. It represents temporal properties that are not identified as a time instant or a time interval, such as the day of the week, if it is holidays or working days, etc. This class has as attributes *value* and its data *type* (i.e., *DayOfWeek* could have as value *Monday* and as type *String*).

To describe the **events** that happen in the system, we reused the *Event* class proposed by SOUPA. In SOUPA, an event is a temporal and spacial thing. Thus, SOUPA provides the *SpatialTemporalThing* class, which is the intersection between *TemporalEntity* and *SpatialThing*. In addition, the *SpatialTemporalEvent* class is defined as the intersection of the Event and SpatialTemporalThing classes. The events in our systems can be a change detected by sensors, or can be an operation executed by a person or automated by the system. Thus, in order to better represent the events of our systems, we refine the SpatialTemporalEvent class in the *DetectedChange* class and the *ExecutedOperation* class. The *DetectedChange* represents a change that has been detected by the devices of the system (e.g., the temperature has increased, presence has been detected, the time goes by, etc.). This class is related with the environment or temporal property whose value has changed (e.g., the temperature of the the kitchen). The *ExecutedOperation* class represents an event produced by the execution of an operation (e.g., switching on the light or playing a song). This class is related with the executed operation (e.g., the switch on operation of the lighting service or the play operation of the multimedia player service) and the arguments used for executing the operation. This class is refined in the *OperationExecutedByPerson* and *AutomaticOperation* classes. The *OperationExecutedByPerson* represents the execution of an operation by a person. This class is related with the person that has executed it by using the *executedBy* relationship. The *AutomaticOperation* represents the execution of an operation by the system.

In Figure 5.1, the class diagram of the ontology is shown. In this diagram, the classes reused from SOUPA are marked with an asterisk. We have extended this ontology with the proposed concepts (those that are not marked with an asterisk) to achieve the representation of all the context information detected in subsection 5.1.1. In addition, these concepts provide a greater semantic richness in order to express how the system provides services to users and how users interact with these services (user behaviour). For instance, we can relate services that are available for users with the locations of the AmI environment in which they are provided; we can also relate actions performed by users with

the service operations that are executed. This information helps the system to study the behaviour of users in more detail to properly adapt itself to it.

It is important to note that this ontology covers the core context information needed for automation purposes. However, other context information specific for a system may be needed. In this case, the ontology can be easily extended with new classes to cover this information.

The Context Model

The **context model** implements the above explained context ontology to semantically describe the context required for properly automating user behaviour patterns. In this model, the context of the system is represented as instances of the classes defined in the ontology.

We specified the context model in the Web Ontology Language (OWL) (Smith *et al.*, 2004). OWL is an ontology markup language that greatly facilitates knowledge automated reasoning and is a W3C standard (more information about OWL can be found in Section 2.3.1). Using OWL, the classes of the ontology are defined by OWL classes, and the context specific of the system is defined by OWL individuals, which are instances of these classes. In OWL, the properties of each class are represented by attributes whose data type is simple. These properties are *DatatypeProperties*. The relationships with other classes are represented by attributes whose data type is a class. These properties are *ObjectProperties*. For instance, a user named Bob is specified as an individual of the Person class whose *ID* DatatypeProperty is *Bob*. Its preferred temperature is specified as an individual of the *Preference* class and added to the *userPreferences* objectProperty (which contains the list of user preferences) of the *Bob* individual. Figure 5.2 shows an example of context model using a graphical tree representation and in OWL format. In this figure, some of the classes and properties of the created context ontology are shown as well as some individuals created as examples.

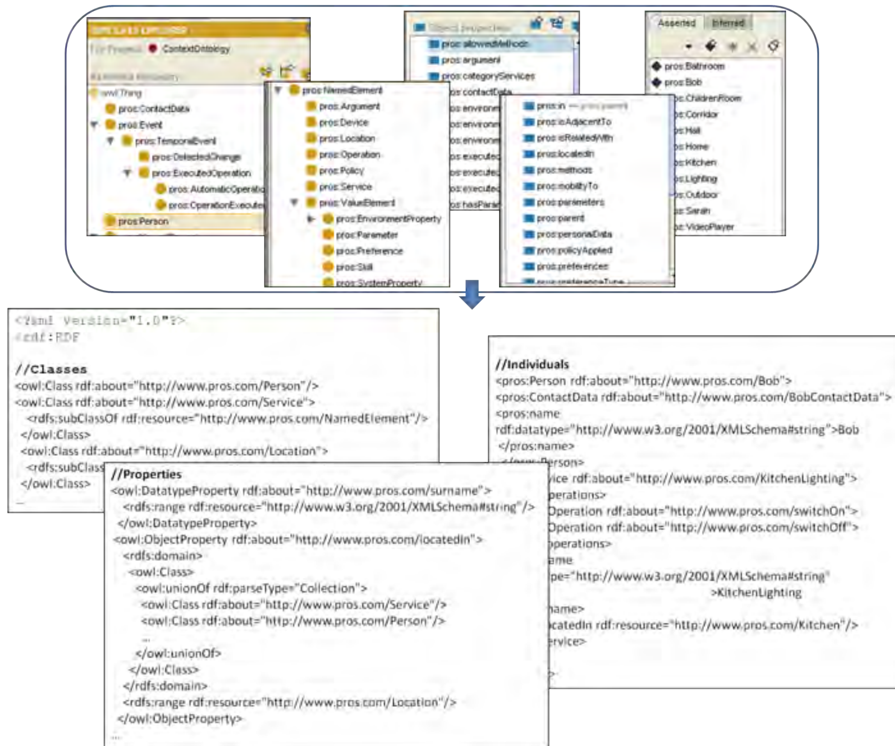

Figure 5.2: An example of a context model shown in a tree representation on the top and in OWL code on the bottom

5.1.4 Tool Support for Creating a Context Model

There are several tools that can be used to create OWL models. Some examples are SWOOP[1], the editor developed by the Model Feature company[2], the OWL Visual Editor of the EMF Ontology Definition Metamodel (EODM)[3] plugin developed upon the eclipse platform (Eclipse, 2011), the SematicWorks tool developed by Altova [4],

[1]http://www.mindswap.org/2004/SWOOP/
[2]http://www.modelfutures.com/
[3]http://wiki.eclipse.org/MDT-EODM
[4]http://www.altova.com/products/semanticworks/semantic_web_rdf_owl_editor.html

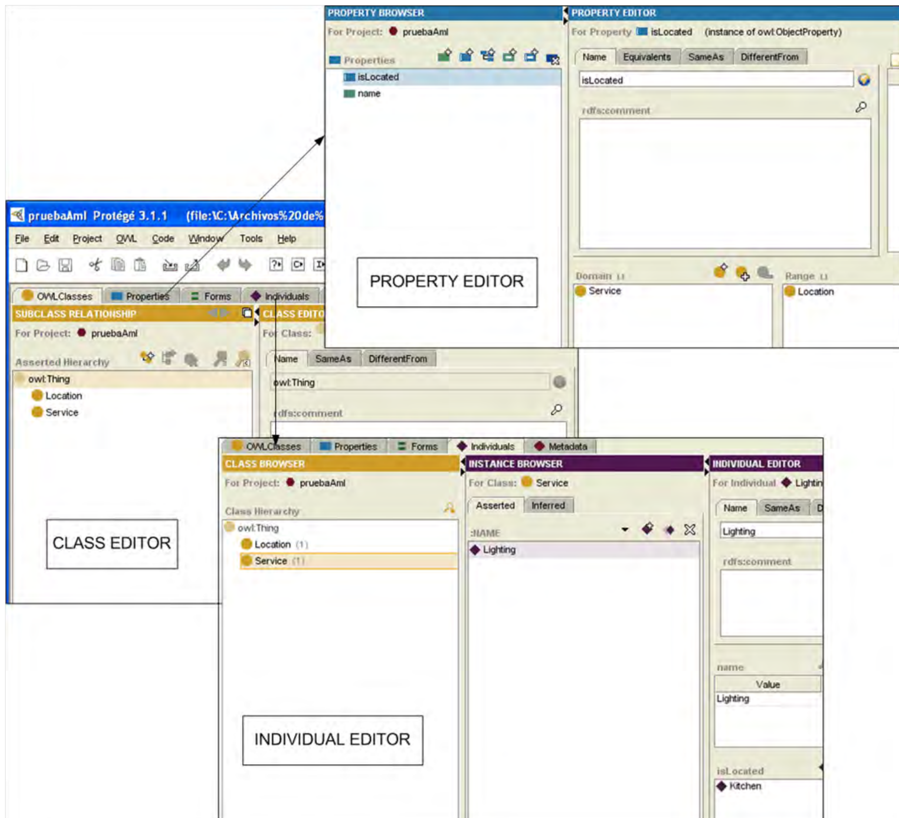

Figure 5.3: Snapshot of the Protégé user interface

or Protégé[5]. Any of these tools can be used to create the OWL context model. We have used the Protégé tool because it is an open source tool that can be freely downloaded from its Web page, and because it provides a very intuitive interface to create ontologies. In addition, there is a great research community that is continuously extending and improving this tool. A proof of that is the International Protégé Conference that is celebrated each year.

Figure 5.3 shows a snapshot of the Protégé tool. In this snapshot

[5] http://protege.stanford.edu/overview/protege-owl.html

the above introduced OWL model is being created. The Protégé user interface is divided in several tabs that provide us with editors to create the different elements of the ontology: classes, properties and individuals.

5.2 Modelling the Behaviour Patterns

A suitable model for specifying the behaviour patterns to be automated is needed. In this model, the tasks that compose each behaviour pattern and how and when they must be executed must be described. In this section, we first introduce some background of the task concept. Next, we revisit the most important models that have been proposed to specify tasks. Finally, we explain the task model proposed in this thesis.

5.2.1 The Task Concept

The *task* concept can be found with different meanings throughout the literature. Some examples of these meanings are the following:

- In common language, a task is a piece of work to be done, especially one done regularly, unwillingly or with difficulty (Card *et al.*, 1983).
- From a computer perspective, a task is an operating system concept which refers to "an execution path through the address space". In other words, a task is a set of program instructions that is loaded in memory. In this field, task is also known as process or job (Silberschatz *et al.*, 2004).
- From a project management perspective, a task is a specific work item to be undertaken that usually results in partial completion of a project deliverable (Westland, 2003).
- From a task analysis perspective, a task is a group of discriminations, decisions and "effector" activities related to each

other by temporal proximity, immediate purpose and a common man-machine output (Miller, 1956).

- From a task modelling perspective, tasks are activities that have to be performed to reach a goal, where a goal is either a desired modification of the state of an application or an attempt to retrieve some information from an application. Thus, tasks can be either logical activities such as *retrieving information about the ambient temperature*, or physical activities such as *switching on the lights* (Paternò, 2001).

In this thesis, we apply task modelling for specifying the behaviour patterns that users want to be automated. Thus, we base on the first and the last definitions to describe the task concept. Taking into account that this thesis is focused on the automation of the described tasks, we consider a task to be:

An activity done regularly that users want to be automated in order to achieve a certain need or goal.

5.2.2 Task Modelling in Software Engineering

Task modelling can be defined as a process in which the tasks to be performed are precisely described detailing the relationships among the identified tasks. In order to describe the tasks to be executed in each behaviour pattern we need a model that deal with two important requirements according to the goals of this thesis (see Section 1.3):

- being intuitive enough for users to facilitate their participation in the modelling of the behaviour pattern tasks; in this way, users' desires and demands can be properly taken into account.

- providing enough expressivity to accurately specify the behaviour pattern tasks in such a way that they can be automated from their specification (by interpreting the model at runtime).

Several models have been proposed to model behaviour, such as Task Model, Use Case Diagram, Interaction Diagram, Business Process

Model, etc. In this work, we decided to use task models because they provide a notation closer to the concepts of user behaviour patterns. In addition, they properly fulfil the above requirements:

- Tasks centre the modelling process around the users' own experiences and goals (Lauesen, 2003), and are well understood by the users (Johnson, 1999).
- Task-based models can be very expressive to describe human-computer interactions (Johnson, 1999) and to do it in a machine understandable way (Limbourg & Vanderdonckt, 2004).

Furthermore, task models have been effectively applied for achieving many goals, such as the following ones (Huang *et al.*, 2008; Limbourg & Vanderdonckt, 2004; Paternò, 2001, 2002; Pribeanu *et al.*, 2001; Sousa *et al.*, 2006):

- Understanding an application domain: since it requires a precise identification of the main activities and their relationships, task modelling helps to clarify many issues that may not be immediately recognised at the beginning.
- Recording the results of interdisciplinary discussions: since many people can be involved in the design of an interactive application (user interface designers, software developers, managers, end-users, experts of the application domain), it is important to have a representation of the activities that can integrate all the requirements raised and that can be understood by all of them;
- Documenting interactive software: the description of how activities are supported by the application is a documentation useful for users, to learn how to use it, and for developers, to have an abstract description of the implementation.
- Designing user interfaces: task models can focus on how humans interact with a particular user interface in a given context of use, possibly interacting with other users at the same time.

- Designing new applications consistent with the user conceptual model: applications designed following a task-based approach are usable and incorporate the user requirements captured in the task model.
- Giving support for the interaction between users and system: task modelling can be very useful for assisting end-users in the execution of tasks through service provisioning and resource allocation.
- Analysing and evaluating usability of an interactive system: task models can be useful to predict the users' performance in reaching their goals or to support analysis of user behaviour to identify usability problems.

Due to such different goals, numerous task model formalisms and methodologies have been developed. Some of the most important are GOMS (John & Kieras, 1996), UAN (Hartson & Gray, 1992), TKS (Johnson *et al.*, 1992), GTA (Veer *et al.*, 1996), HTA (Shepherd, 2001), or CTT (Paternò, 2002). These works show the growing usage of task modelling and its remarkable results and possibilities to model user interaction with the system. However, they have not been proposed with the goal of automating this interaction (i.e., with the goal of automating user behaviour patterns). Therefore, the proposed task models do not provide enough expressivity either to specify whole behaviour patterns (such as specific relationships between tasks, context awareness, etc.) or to accurately describe them to be directly executed.

In this thesis, we base on Hierarchical Task Analysis (HTA), which was date back to the late sixties and has proved to be successful, as can be seen from its application in a big number of projects. The basic idea of HTA is to describe the set of activities to be considered logically structured in different levels as a task hierarchy. HTA views tasks in a more abstract sense, as a set of interlinked goals, resources and constraints (Shepherd, 2001). It allows us to represent tasks starting from more general tasks and ending with more specific ones. Thus, tasks can be described at different levels of abstraction and detail.

In addition, HTA facilitates the representation of certain dependencies defining the order of execution. Often, such dependencies are described by a set of temporal equations, using predefined temporal operators, which allow us to ordering and detailing how the tasks must be executed. To define them, we base on the temporal relationships provided by ConcurTaskTrees (CTT) (Paternò, 2002), which define a rich set of temporal operators that can be used to temporally order the tasks. These temporal operators are focused in the user interface design, therefore, we extend them to properly allow the automation of tasks. Thus, our task model can be seen as an extension of the combination of HTA and the CTT temporal operators.

5.2.3 A Context-adaptive Task Model

To specify the behaviour patterns that users want to be automated, we propose a context-adaptive task model. In this model, the behaviour patterns are specified by using context information captured in the context model. Specifically, a behaviour pattern is described by: the context conditions whose fulfilment enables the pattern and the set of tasks that compose the pattern and that must be carried out to perform it. These tasks are also specified according to context and are related between them using temporal relationships so that the tasks can be properly executed. Thus, the automation of the behaviour patterns is adaptive to context, which is essential to achieve that the automation is performed unobtrusively. For this reason, we refer to the proposed model as a context-adaptive task model.

This model attempts to be comprehensible enough so that people without high-level training can understand it; but also attempts to be expressive and precise enough so that the model can be directly interpreted to automate the described behaviour patterns.

As we have said, the proposed context-adaptive task model is based on the *Hierarchical Task Analysis (HTA) technique* (Shepherd, 2001), which hierarchically refines more general tasks into more specific tasks. Next, we detail the model using real examples of behaviour patterns identified in the performed smart home case studies. These case studies,

which will be explained in Chapter 8, attempt to improve users' lives and saving energy resources by automating users' daily tasks. To detail the model, we present how the behaviour patterns are represented, how they are refined, how we achieve that they are adaptive to context, and finally, we define the task model syntax by describing its metamodel.

Representation of behaviour patterns

Historically, HTA techniques have used tabular or graphical representations to specify tasks. In tabular representations, tasks are textually specified by means of tables. Each task is represented by a row of a table. Task refinements are indicated by using for instance a specific task numeration (e.g., the Task 1.1 is a subtask of the Task 1). In graphical representations, tasks are specified by a tree whose nodes represent tasks and whose branches represent task refinements. Tasks in a level of the tree constitute the subtasks of the task in the upper level to which they are directly connected.

In this work, we use a graphical representation because we consider it to be more intuitive and easier to manage by analysts than tabular descriptions. Currently, there are several notations that allow us to graphical represent tasks. Some of the most important notations are CTT (Paternò, 2002), GTA (Veer *et al.*, 1996) or TKS (Johnson *et al.*, 1992). These notations are usually focused on design interfaces and they provide a myriad of notations to represent tasks of different types as well as the refinement of them. They are very powerful tools to achieve the purpose of facilitating the design of software. However, our model only has to specify tasks to be automated; therefore, we can define a simpler notation to facilitate the comprehension of the specified behaviour patterns by users.

Thus, we based on a very simple notation (Valderas, 2008) in which the task hierarchy is defined from ellipses and lines between them. Each ellipse represents a task. Ellipses in upper levels in the taxonomy indicate more general tasks. Ellipses in lower levels indicate more specific tasks.

Since a behaviour pattern represents a set of tasks that are

performed to achieve a common goal, we propose defining a **task hierarchy** for each behaviour pattern. In this task hierarchy, the root task represents a behaviour pattern. As examples, Figure 5.4 shows the modelling of three behaviour patterns named *WakingUp*, which encapsulates the tasks to be performed for waking up users, *Leaving*, which encapsulates the tasks to be performed when users leave home, and *StormSecurity*, which encapsulates the tasks to be performed when it starts raining. These behaviour patterns are real examples identified for the developed case studies (see Chapter 8). Each one of the behaviour patterns has an associated context situation that defines the context conditions whose fulfilment enables the execution of the behaviour pattern. It is represented by a note linked to the root task. For instance, the behaviour pattern *WakingUp* is associated with the context situation *CurrentTime=7:50 and WorkingDay=true*, i.e., at 7:50 in working days.

The root task has also a priority (High, Medium, and Low) to establish the priority of execution of the pattern in case several patterns are enabled at the same time. To make the model notation easier, this is not graphically shown. From here, the elements that are not explicitly said to be graphically represented in the model, it is because they are not. These elements are not represented for not overloading the model and facilitating its comprehension.

This root task can be broken down into *composite tasks* (which are intermediate tasks) and/or *system tasks* (which are leaf tasks). Composite tasks are used for grouping subtasks that share a common behaviour or goal, such as the *lighting* task of the *WakingUp* pattern or the *manage home resources* of the *Leaving* pattern. System tasks represent atomic tasks that have to be performed by the system, such as *turn on bathroom heating* or *turn on the radio* tasks of the *WakingUp* pattern. Hence, each system task has to be related to a pervasive service that can carry it out. The relation is established by means of the name of the service and the name of its corresponding operation. Note that a task represents a goal for accomplishing a user need, while a service is the performance of this goal. For instance, the *turn on the radio* system task is associated to the *turnOn* operation provided by

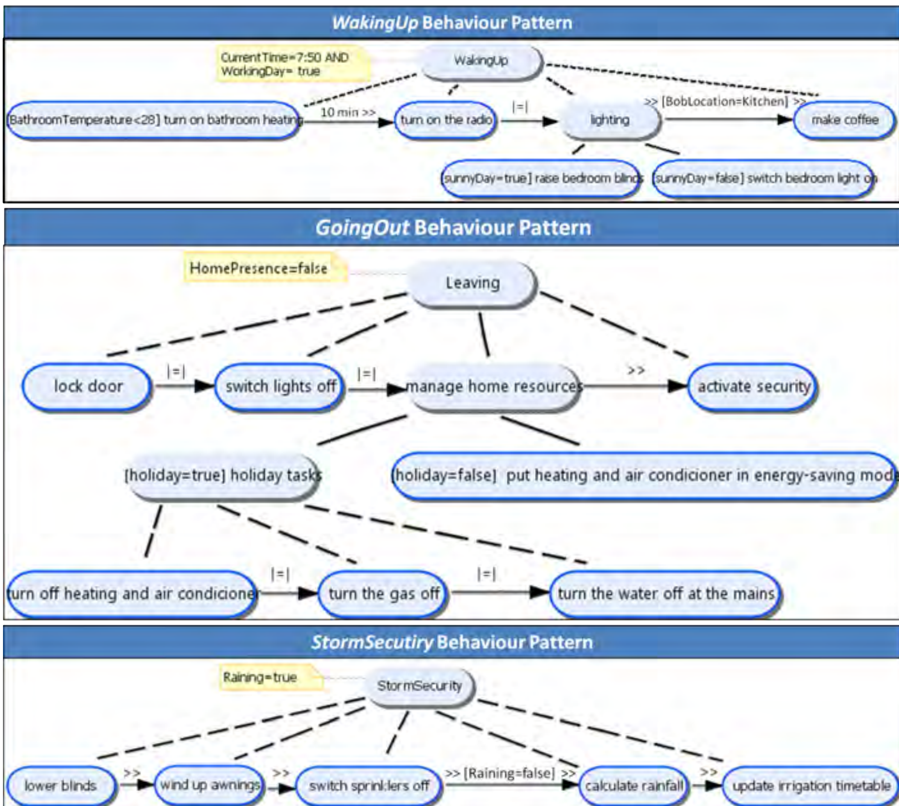

Figure 5.4: Example of behaviour pattern modelling (graphical representation)

the *Radio* pervasive service, which executes this action by interacting with the radio.

Both system and composite tasks can have a context precondition (which is shown between brackets before the name of the task). It defines the context conditions that must be fulfilled so that a task is performed. If the precondition is not fulfilled, the task will not be executed and the system will pass to execute the next task. For instance, the *turn on bathroom heating* task of the *WakingUp* pattern is only executed when its precondition *BathroomTemperature<28* is fulfilled.

In addition, tasks inherit the context preconditions of their parent task.

Also, system tasks can have input and output parameters. The input parameters correspond to the parameters that the service related to the task may require to be executed. For instance, the *turn on the radio* task of the *WakingUp* pattern needs the radio channel input parameter. The output parameter corresponds to the return value that the service related to the task may have. This parameter can be used as an input parameter in next tasks of the behaviour pattern. For instance, the *calculate rainfall* task of the *StormSecurity* pattern returns the rainfall; value that is needed by the *update irrigation timetable* task.

In addition, each task is defined by a task name (which is the text that explains the goal of the task in a user comprehensible way and that is shown inside the ellipse) and an internal task ID (which is a unique identifier).

Refinement of Tasks

In order to refine a behaviour pattern or a composite task into simpler tasks, we propose two types of refinement: the *exclusive* refinement and the *temporal* refinement. A refinement is represented by a line that connects a parent task with a subtask that is in the immediately lower hierarchy level.

The *exclusive refinement* is represented by a solid line between a parent task and a subtask. Using this refinement, a task is decomposed into a set of subtasks, in such a way that only one subtask will be executed (disabling the others). This refinement must be used when to achieve a user goal, a different task must be carried out depending on context. For instance, in the *WakingUp* pattern, the *lighting* task is refined into two subtasks by exclusive refinements because only one of these tasks must be executed depending on whether it is a sunny day or not.

The *temporal refinement* is represented by a dashed line between a parent task and a subtask. Using this refinement, a task is decomposed into a set of subtasks that must be executed following a certain

order. This refinement must be used when the user goals involved in a task constitute a whole activity that can be partitioned in several coordinated parts. Thus, in order to complete the whole activity, all the parts in which it is partitioned must be completed. This is known in HTA as a plan (Shepherd, 2001) (the plan indicates the way in which subtasks must be performed). To define this plan, we propose the use of **temporal operators** to link the subtasks of a parent task. The temporal operator between two tasks will also be applied to their child tasks.

We base the definition of these operators on CTT (Paternò, 2002), which provides one of the richest sets of temporal operators. However, our model has to be prepared to be executed. For this reason, every operator must indicate precisely which task must be executed and when it must be executed. This make us dismiss some operators, such as the optional one, for not fulfilling these requirements. In addition, in this work we focus on supporting stateless tasks, which are the most usual and required tasks in behaviour pattern automation (as shown in the developed case studies, in which tasks whose state has to be managed have not been detected; see Chapter 8). This make us dismiss operators that require to store the task state, such as $[<$ or $| <$. Support for tasks with state will be dealt with in further work (see more detail in Chapter 9).

In addition, tasks in a behaviour pattern may require or provide information from and to any task of the pattern. This information passing is supported by the input and output tasks parameters; for this reason, we do not use the CTT temporal operator for information passing, which restricts this action to two consecutive tasks. Therefore, we use the following CTT operators:

- $T1 >> T2$, enabling: the *T2* task is triggered when the *T1* task finishes. For instance, the task of the *Leaving* pattern for activating security is only triggered when the tasks for managing the home resources have been performed.
- T1| = |T2, task independence: T1 and T2 can be performed in any order. For instance, the execution order of the *turn on the*

radio and *lighting* tasks of the *WakingUp* pattern is not relevant because both of them are used to wake up the user.

To properly capture the pervasive system automation requirements, we extend the enabling operator obtaining two additional ones:

- T1 $t >>$ T2, enabling after t minutes: executed T1, T2 is enabled after t minutes. For instance, in the *WakingUp* pattern, 10 minutes after the *turn on bathroom heating* task finishes, the *turn on the radio* task is enabled. Thus, the bathroom is warm when the user enters to take a shower.
- $T1 >> [c] >> T2$, enabling when c is fulfilled: after executing T1, T2 is enabled when the condition c is fulfilled. For instance, in the *StormSecurity* pattern, after the *switch sprinklers off* task has finished, the *calculate rainfall* task is not enabled until it stops raining (*raining=false*). For security, this task can have also an associated temporal restriction, just in case the condition is never satisfied or is satisfied when the execution of the task is not needed any more. Thus, if the time specified in the temporal restriction goes off before the context condition is satisfied, T2 is disabled.

We depict temporal operators by attaching them to an arrow that connects the two related subtasks. These subtasks have to have the same parent task and it has to be refined by means of temporal refinements.

When should the refinement stop?

One of the most difficult aspects in hierarchical task modelling is the following: when we know that the decomposition of tasks is finished. This problem was already discussed at the beginnings of HTA (Annett & Duncan, 1967). HTA was initially proposed to be a general method for examining work. Annet and Duncan suggested a stopping rule known as the PxC rule: analysts should finish the decomposition of a task when the probability of failure (of performing an inadequate decomposition) (P) multiplied (x) by the cost of failure (C) surpasses

a specific predefined level of acceptance. In this case, analysts should estimate P, should estimate C and should establish the predefined level of acceptance. This is not always easy, as the authors Annet and Duncan admit.

Other more recent approaches such as (Shepherd, 1993) or (Ormerod & Shepherd, 2003) propose a rule based on the analysis of goals. They propose to finish the refinement when the goal that must be achieved by a task is of low level. They introduce different types of low-level goals in order to facilitate their identification. For instance, they explain that a low-level goal can be a goal that only implies an action which changes the state of the system or those that only implies an observation of the state of the system.

In this thesis, we are inspired by this last approximation. However, we want to provide a more practical rule for the purpose of identifying the set of the tasks that represent the user needs. To do this, we must analyse the definition of task that is provided above. We consider a task to be an activity done regularly that users want to be automated to achieve a certain need or goal. According to this definition, a task represents an activity that must be automated by the system. Thus, we stop decomposing tasks when a subtask constitutes an atomic action: if the system services are available before the task modelling, decomposition stops when the system can directly execute the task by using an operation of one of its services; if the system services are developed after the task modelling, decomposition stops when the granularity of the task is considered suitable and feasible to be implemented in a service operation. We have called these atomic actions as *system tasks*.

Context Adaptivity

Note that context adaptivity is achieved by means of: the context conditions specified in the context situation that enables the execution of a pattern, task preconditions, and relationships between tasks. In order to specify these conditions, we use a logical expression. This expression combines any number of basic expressions linked by the

following logical connectives: and (AND), or (OR), equalities (=), inequalities (!=) and greater (>), or less than (<). The context properties used in these expressions have to be specified in the context model.

The input parameters of system tasks can be context properties and then can be also used for adapting the task execution to context. Using them, the execution of the service related to the task is executed according to the values of the context parameters.

To refer to a context property, the name of the context property specified in the context model and the name of the individual to which this property belongs have to be indicated (e.g., a context property could be specified as the *value* property of the *Temperature* individual). These two elements identify the needed context property in the context model, which allows us to search for its value in the context model.

The Task Model Metamodel

To formalize the elements that can appear in the task model (i.e., its abstract syntax), we define its metamodel. Thus, the task model is unambiguous at the syntactic level. The class diagram of this metamodel is shown in Figure 5.5.

The main element of the task model metamodel is the *Task* class. Tasks have a *name* and an *ID*. There are two types of task: *System Task* and *Composite Task*.

System Tasks represent the tasks that have to be carried out by the system. Thus, each system task has a service name, a service method name and a set of arguments (each one of them has a name, a type and a value) that may be needed for executing the indicated method. In addition, a system task can also have an output argument, which indicates the result of executing the corresponding method.

Composite Tasks are refined into other tasks (system or composite). The refinement is performed by a *Refinement*. A refinement can only refine one task into other task, i.e., a task can only have one parent task in the task taxonomy, and for each new child task a new refinement must

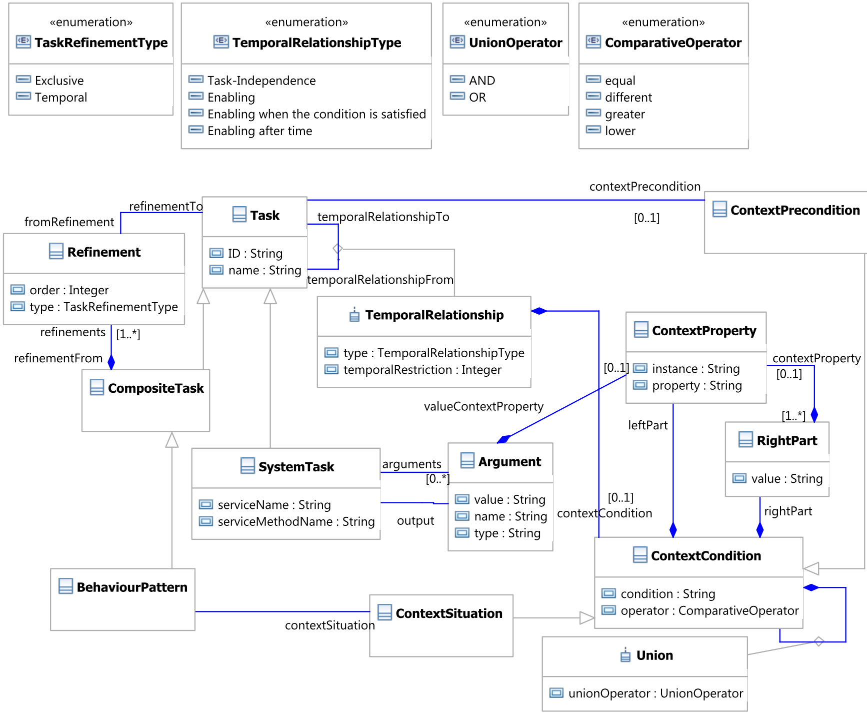

Figure 5.5: Overview of the task model metamodel

be created. A refinement has an *order* to indicate the order in which the tasks has to be checked, and a type, which can be *Exclusive Refinement* or *Temporal Refinement* as the enumeration indicates.

When a composite task has been refined by using the temporal refinement, its subtasks are related by means of a *Temporal Relationship*. Two tasks can only be related by one temporal relationship. Temporal Relationships also present the attribute *type* which indicates its type according to the enumeration *TemporalRelationshipType*: Task-Independence, Enabling, Enabling when c is satisfied, Enabling after time.

A *Behaviour Pattern* is defined as a type of composite task. Each

Behaviour Pattern has to be related with a *Context Situation* element, which is a *Context Condition* that must be fulfilled for activating the behaviour pattern. Tasks can also have a *Context Precondition*, which is also defined as a type of Context condition. In addition, temporal relationships of the type *Enabling when c is satisfied*, has to be related with a context condition that must be satisfied so that the system continue executing the next task. A context condition can be any number of basic expressions (equalities (=), inequalities (!=), and more than (>), or less than (<)) that have a left part and a right part. The left part is always a context property while the right part can be a context property or a value. Thus, in the basic expressions, a context property can be compared with another context property or with a value. In addition, a condition can be composed by basic expressions linked by the connectives: and (*AND*), or (*OR*). A *Context Property* has the name of the instance and the name of the instance property (which are needed for searching for the context property in the context model). Thus, an expression can be formed by two context properties or by a context property and a certain value. Since the task model must be understood by end-users, we do not give support for forming more complicated context conditions.

Furthermore, the following constraints need to be considered in order to correctly define a task model:

- Constraint 1. Two tasks cannot have the same ID. In OCL:

  ```
  Context Task

  Inv: self.allInstances -> forAll(t1, t2 | t1<>t2
    Implies t1.ID <> t2.ID)
  ```

- Constraint 2. Temporal Relationships can only be used to relate tasks that are both child of a same parent task which is refined using the temporal refinement. In OCL:

```
Context TemporalRelationship
Inv:
self.temporalRelationshipFrom.
 fromRefinement.type='Temporal'
AND
self.temporalRelationshipTo.
 fromRefinement.type= 'Temporal'
AND
self.temporalRelationshipFrom.
     fromRefinement.refinementFrom=
self.temporalRelationshipTo.
 fromRefinement.refinementFrom
```

- Constraint 3. A refinement cannot refine a task into a behaviour pattern. In OCL:

```
Context Refinement

Inv: not self.refinementTo.isInstanceOf
('BehaviourPattern')
```

- Constraint 4. When a task is refined only one type of refinement can be used to obtain child tasks. In OCL:

```
Context CompositeTask

Inv: self.refinements -> forAll(t1, t2 |
 t1.type = t2.type)
```

- Constraint 5. Each task refined by using a temporal refinement, has to be related using a temporal relationship. In OCL:

```
Context CompositeTask
Inv:
self.refinements -> forAll(t1 | t1.type =
```

```
'Structural')

xor

self.refinements -> forAll(t1 | t1.type =
'Temporal')
and
self.refinements-> forAll(r1 | r1.refinementTo.
temporalRelationshipFrom<>null
or r1.refinementTo.temporalRelationshipTo<>null)
```

- Constraint 6. A behaviour pattern cannot have a context precondition. In OCL:

```
Context BehaviourPattern
Inv: self.contextPrecondition=null
```

- Constraint 7. Each temporal relationship whose type is *Enabling when the condition is satisfied* has to have a context condition. In OCL:

```
Context TemporalRelationship
Inv:
self.allInstances -> select(t1 | t1.type='Enabling
when the condition is satisfied')-> forAll(t1
| t1.contextCondition<>null)
```

- Constraint 8. Each temporal relationship whose type is *Enabling after time* has to have a temporal restriction with a time greater than 0. In OCL:

```
Context TemporalRelationship
Inv:
self.allInstances -> select(t1 |
```

```
t1.type='Enabling after time')
-> forAll(t1| t1.temporalRestriction>0)
```

5.2.4 Tool support

In order to support the graphical specification of the proposed context-adaptive task model, we have developed a graphical tool based on the tool developed in (Valderas, 2008), which defines the notation on which ours is based.

The technology used to develop this tool is based on the Eclipse platform (Eclipse, 2011). Specifically, the following Eclipse plugins have been used:

- The Eclipse Modelling Framework (EMF) project provides us with a modelling framework and code generation facilities for building tools and other applications based on a structured data model. The core of this framework includes both a meta model (Ecore) for describing models and runtime support for managing models, including change notification, persistence support with default XMI serialization, and a very efficient reflective API for manipulating EMF objects generically. Ecore is an implementation of the Essential Meta-Object Facilities (EMOF), which is a subset of the standard MOF 2.0 [MOF] proposed by the Object Management Group (OMG) for describing meta-models.

  Furthermore, EMF provides mechanisms to generate Java files from an Ecore metamodel. These files implement the different elements of the metamodel (by means of Java classes) providing support to create instances of them at runtime (i.e., objects of a Java class that represent elements of the metamodel) as well as to manage them.

- The Graphical Editing Framework (GEF) project allows developers to take an existing application model and quickly create a

rich graphical editor. Basically, GEF provides an infrastructure for developing graphical editors by following the pattern model-view-controller (MVC). GEF itself provides support to develop the controller part. In order to develop the model and view parts GEF does not force to use specific libraries. However, the most common way of using GEF is together with Draw2D for the view part and EMF for the model part.

- The Eclipse Graphical Modeling Framework (GMF) project provides us with a generative component and a runtime infrastructure for developing graphical editors. GMF allows us to declaratively describe the different associations among elements of a model and their visual representation by means of models. From these models GMF automatically generates a graphical editor (view) implemented by means of GEF. This graphical editor provides support for creating, modifying and deleting (controller) each visual representation. The use of GMF provides us with an abstract way of developing graphical editors, without the need of considering the technological aspects introduced by GEF.

Using these plugins, the tool, which is shown in Figure 5.6, has been performed following the next steps. First, we have specified the metamodel of the context adaptive task model described in the previous subsection (see Figure 5.5) in Ecore format (see Figure A.3). Next, we have used the facilities provided by EMF in order to automatically generate from the Ecore metamodel a set of Java classes that provide support to manage instances of the metamodel elements at runtime. These Java classes constitute the model part of model-view-controller architecture in which the graphic editor is implemented. Finally, the view and controller parts of the tool have been implemented using the facilities provided by GMF.

It is important to note that using this tool, the proposed task model not only can be graphically visualized and edited, but also, is stored in XMI (XML Metadata Interchange), which is a machine-processable language. Figure 5.7 shows part of the XMI representation of the *WakingUp* pattern (see Figure 5.4), where the properties of the *turn*

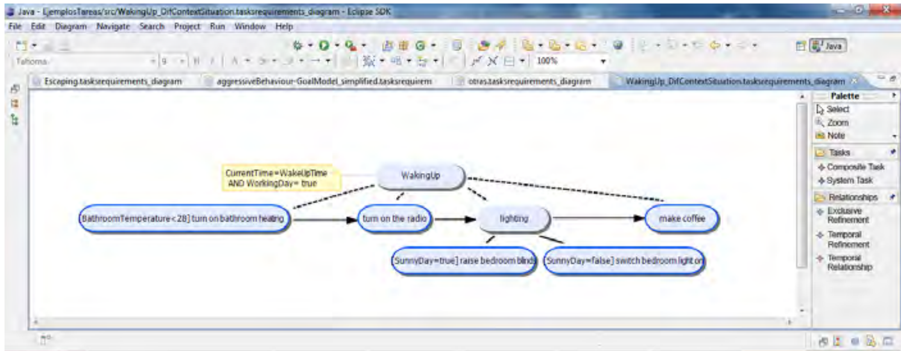

Figure 5.6: Snapshot of the behaviour pattern modelling tool

on bathroom heating and *turn on the radio* tasks are shown.

```
<?xml version="1.0" encoding="UTF-8"?>
<org.pros:TaskModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
...
<Task xsi:type="org.pros:SystemTask" name="turn on bathroom heating"
ID="WakingUp_TOBAHC" refinementFrom="//@refinement.0">
       <ContextPrecondition ContextPreconditionString="BathroomTemperature<28">
       <TemporalRelationship type="t >>" temporalRestriction minutes=10
       TemporalRelationshipTo="//@Task.2" TemporalRelationshipFrom="//@Task.1"/>
    <service name="AirHeatingConditioner" method="switchOn">
</Task>
<Task xsi:type="org.pros:SystemTask" name="turn on the radio" ID="WakingUp_TOR"
refinementFrom="//@refinement.1">
        <TemporalRelationship type="|=|"
       TemporalRelationshipTo="//@Task.3" TemporalRelationshipFrom="//@Task.2"/>
    <service name="Radio" method="turnOn">
</Task>
...
</org.pros:TaskModel>
```

Figure 5.7: Part of the XMI representation of the WakingUp behaviour pattern

In addition, this tool also provides model-based validations in the task metamodel to ensure that the specified behaviour patterns are valid prior to their construction. This allows automatically validate a created task model. These validations check the constraints described in the previous section.

5.3 Conclusions

In this chapter, we have studied some background of context and task modelling to propose two suitable models for specifying the behaviour patterns that users want to be automated: an ontology-based context model and a graphical context-adaptive task model.

We have shown that these models meet the requirements needed for properly achieving the automation of user behaviour patterns: 1) they are intuitive enough to be user comprehensible, which facilitates user participation in the specification of the behaviour patterns; 2) accurate enough to provide all the needed information to automate the specified behaviour patterns; and 3) machine-interpretable to be managed at runtime.

Thus, the proposed models are design models, which allow us to specify the behaviour patterns at a high level of abstraction, but also are executable models, which allow us to execute the behaviour patterns and evolve them when needed by directly managing the models at runtime.

Chapter 6

Automating User Behaviour Patterns

Smart environments are physical environments that are richly and invisibly interwoven with sensors and actuators embedded seamlessly in the everyday objects of our lives, and connected through a continuous network (Weiser, 1991). In order to control these sensors and actuators, smart environments provide us with services that function invisibly and unobtrusively in the background with the final goal of freeing people to a large extent from tedious routine tasks (Mattern, 2001, 2005). This is one of the main challenges of this thesis: automating routine tasks, also known as behaviour patterns.

To achieve this challenge, this thesis applies the guidelines provided by Model Driven Engineering (MDE) to raise the level of abstraction in program specification and increase automation in program development. In the previous chapter, we have explained two models to specify at a high level of abstraction the behaviour patterns that users want to be automated. These models not only provide abstract concepts that facilitate the participation of end-users in the model specification, but

they also are machine-processable and precise-enough to be used as executable models. In this chapter, we focus on explaining a software infrastructure that directly executes these models to automate the specified behaviour patterns in the opportune context.

Before explaining this software infrastructure, we first study the requirements that it must satisfy for properly achieving the automation of user behaviour patterns (Section 6.1). Next, we define the process that must be followed for automating behaviour patterns (Section 6.2). This process is designed according to the requirements previously identified. In the description of this process, the software components that the software infrastructure must provide are identified.

We next explain the software infrastructure and describe in detail its software components (Section 6.3). This infrastructure automates behaviour patterns in the opportune context by following the automation process described and satisfying the requirements identified. In order to explain this infrastructure, we first introduce its software components in an abstract way by explaining their functionality without considering technological details. Afterwards, we describe how these components are implemented in order to support the needed functionality.

Finally, we explain the conclusions of the chapter (Section 6.4).

6.1 Requirements for Automating Behaviour Patterns

The task and context models explained in Chapter 5 specify the behaviour patterns that users want to be automated. Since these models are machine-processable and precise-enough, they can be directly used for automating the behaviour patterns. Two strategies can be used in order to achieve this (see Chapter 2): code generation and model interpretation.

To select the most suitable strategy, it is important to consider that users' behaviour may change over time. Therefore, the specified

behaviour patterns may need to be changed after system deployment to adapt to user behaviour changes; otherwise, system may become useless, obsolete, or even intrusive. Thus, one of the main requirements in the automation of the specified behaviour patterns is that they must be easily evolved after system deployment.

Considering the advantages of code generation and model interpretation, the latter provides more benefits regarding evolution. Model interpretation enables faster changes because changes in the models do not require an explicit regeneration, rebuild, retest, and redeploy phases. In addition, this strategy facilitates to perform dynamic changes because the models are interpreted at runtime. This makes possible to change the model and consequenly the running application without stopping it.

Furthermore, by automating the behaviour patterns using an engine that directly interprets the models at runtime we achieve that these models are the primary means to understand, interact with, and modify the behaviour patterns. Thus, we achieve two important goals. First of all, we ensure that the behaviour patterns are automated as specified in the models because they are the only representation of the behaviour patterns; consequently, we also ensure that the system automates exactly what the users want because the behaviour patterns described in the models are specified and validated with user participation. Secondly, we allow the behaviour patterns to be evolved by simply updating the models. This achieves the behaviour pattern evolution is performed by using the own modelling language, i.e., using the high level concepts defined in the metamodel of the task model and the context ontology. Detailed information about how this evolution is addressed will be explained in Chapter 7.

6.2 Behaviour Patterns' Automation Process

To execute the behaviour patterns as specified in the models, two important things have to be considered: 1) the opportune context in which each behaviour pattern has to be triggered, and 2) the tasks to be

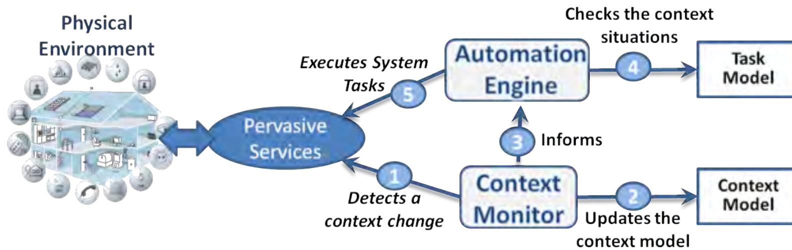

Figure 6.1: Process for Automating User Behaviour Patterns

executed in each behaviour pattern taking into account their execution order and the context in which they have to be executed.

The context in which each behaviour pattern has to be triggered is specified in the task model as a context situation related to each behaviour pattern. When this context situation is fulfilled, the related behaviour pattern has to be carried out. To check whether or not a context situation is fulfilled, the used context, which is stored in the context model, has to be continually updated according to the context changes.

When a context change is produced, the context situations related to this change have to be checked. If a context situation is satisfied, its related behaviour pattern specified in the task model is carried out by executing their tasks in the adequate order and context.

All this information, which is needed to carry out the behaviour patterns, is specified in the task and context models. These models are directly interpreted at runtime by an engine to automate the behaviour patterns as specified. Specifically, the process for automating the specified user behaviour patterns consists of the following steps (see Figure 6.1):

Detecting context changes: the execution of the specified behaviour patterns depends on context information such as time, temperature, light intensity, etc. This context information is continuously changing. Note that context changes are physically detected

by sensors. These sensors are controlled by pervasive services provided by the smart environment. Thus, to capture context changes, these pervasive services are continuously monitored. To do this, a context monitor is needed. For instance, the execution of the *WakingUp* behaviour pattern described in Section 5.2.3 depends on the current time and date, the bathroom temperature, the outside light intensity and the location where the user is. For controlling each one of these properties, the environment provides a service. All of them are monitored to check if the value of any context property has changed.

Updating the context model: the context information on which the behaviour patterns depend is managed using the context model; therefore, when a context change is detected by the context monitor, it updates the context model to reflect the change. To do this, mechanisms for managing the context model at runtime are needed. For example, since the *WakingUp* behaviour pattern depends on the outside light intensity, the context model has the *sunnyDay* property which is an instance of the environment property class (see Section 5.1). When the outside light intensity changes, the value of this property is updated.

Informing the automation engine: an automation engine is needed to automate the behaviour patterns specified in the models by interpreting them at runtime. This engine automates each behaviour pattern when its context situation is fulfilled. For this reason, when a context change is produced, the context monitor informs the engine about this change.

Checking context situations: when the engine is informed about a context change, it analyses the task model in order to check if there is any context situation that depends on the updated context information. To do this, mechanisms that allow the management of the task model at runtime are needed. For instance, when time changes, the *currentTime* property is updated and the engine is notified. Then, it checks whether or not there is any context

situation specified in the task model that is satisfied with the new value of the property.

Executing behaviour patterns: The behaviour patterns whose context situation is satisfied are executed according to their priority. To execute a behaviour pattern, the engine executes the system tasks of the corresponding pattern according to their refinements, their context conditions in the current context and their temporal relationships specified among them. To obtain this information, the engine uses the task model management mechanisms to interpret the task model at runtime. For instance, if the context situation of the *WakingUp* pattern is fulfilled (see Figure 5.4), if the temperature of the bathroom is less than 28 °C, the bathroom heating is turned on, ten minutes later, the radio is turned on. Then, if it is sunny day, the bedroom blinds are raised; otherwise, the lights are switched on. Finally, when the user enters in the kitchen, the system makes a coffee. To execute each system task the engine uses the pervasive service associated to that task.

6.3 Software Infrastructure

In order to carry out the automation process, we provide a software infrastructure that automates the behaviour patterns in the opportune context by using the models at runtime. This infrastructure allows that the only task for achieving the automation of behaviour patterns is specifying them using the proposed models.

Next, we first explain this infrastructure at the conceptual level, and then we explain how we have developed it giving technological details.

6.3.1 Components of the Software Infrastructure

This section presents a technology-independent description of the software infrastructure that carries out the automation process above described. This infrastructure is built by a set of reusable and modular components that collaborate to automate the behaviour patterns as

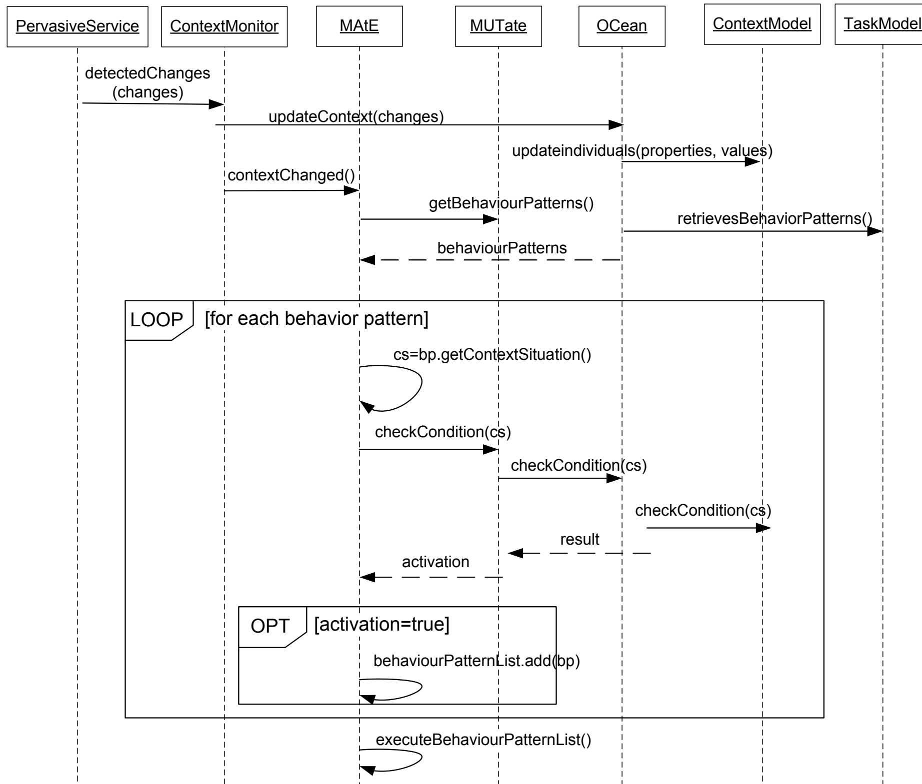

Figure 6.2: Automating User Behaviour Patterns

specified in the models. These components are: pervasive services, the mechanisms for managing the models at runtime, a context monitor and a Model-based Automation Engine. Figure 6.2 shows how these components interact among them to perform the automation process. More details about these components are given below:

Pervasive services. Every smart environment provides pervasive services to control the devices of the environment and sense context information. Specifically, we consider a service to be an entity that provides a coherent set of functionality which is

described in terms of atomic operations (or methods). These operations allow the system to control the devices of the environment in order to change context and/or sense it. Our approach uses these pervasive services in order to perform the tasks of the behaviour patterns specified in the models and to sense context changes.

Mechanisms for managing the models at runtime. In order to manage the context model and the task model (see Chapter 5) at runtime, we have defined Ontology-based Context model management mechanisms (OCean) and Model-based User Task management mechanisms (MUTate).

- OCean: The context on which the behaviour patterns depend is specified in the context model as OWL individuals. Thus, in order to manage these individuals, a set of Ontology-based Context model management mechanisms (OCean) is needed. OCean allows, for instance, updating the individuals of the context model, creating a new individual (e.g., the *idealTemperature* individual of the *Preference* ontology class), reading its properties or modifying them when needed, etc.
- MUTate: In order to support the management of the task model, a set of Model-Based User Task management mechanisms (MUTate) is needed. MUTate allows, for instance, searching for a behaviour pattern that have to be executed; obtaining its related context situation, adding new tasks to a pattern; creating a new pattern; etc.

OCean and MUTate determine the vocabularies and calling conventions used to access the models. Specifically, they provide the same vocabulary defined in the context ontology and the task model metamodel, respectively. It is important to note that, in this way, they provide high-level abstraction mechanisms that facilitate the interaction with the models without the need to stop the system. Both, OCean and MUTate are needed in order

to achieve the automation and evolution of the specified user behaviour patterns.

Context monitor. In order to monitor context changes and update the context model accordingly, a context monitor is used. Context changes are physically detected by the pervasive services that control the system devices. Thus, in order to capture context changes, the monitor is continuously monitoring the execution of the pervasive services. When a change in context is detected by a pervasive service, the context monitor updates the context model accordingly. Note that this update must be performed at runtime. To do this, the context monitor uses OCean. Once the context model has been updated, the context monitor informs the automation engine about this change.

Model-based Automation Engine (MAtE). In order to carry out the behaviour patterns as specified in the models, a Model-based Automation Engine (MAtE) is needed. When a context change is produced, MAtE checks the context situations specified in the task model. If a context situation is fulfilled, MAtE automates the related behaviour pattern. To perform these steps, MAtE uses MUTate. To automate a behaviour pattern, MAtE executes its system tasks by taking into account the current context, their relationships and their refinements. Each system task is executed by MAtE using the pervasive service related to it.

6.3.2 Implementation of the Software Infrastructure

As explained above, the provided software infrastructure is composed by: 1) the pervasive services that control the devices of the environment, 2) OCean, the set of mechanisms that allows the context model to be managed at runtime, 3) MUTate, the set of mechanisms that allows the task model to be managed at runtime, 4) a context monitor that manages the context information, and 5) MAtE, the automation engine that is in charge of executing the behaviour patterns.

All these components are developed using Java/OSGi technology.

This technology is more and more used for developing pervasive computing systems due to the numerous important benefits that it provides (see Section 2.4). Using this technology, we achieve that the software infrastructure is operative system independent and can be dynamically constructed from reusable and collaborative components, which are known in the OSGi terminology as bundles.

Thus, the infrastructure is developed to be run in an OSGi service platform. An OSGi service platform is an instantiation of a Java virtual machine, an OSGi framework, and a set of bundles.

The OSGi framework runs on top of a Java virtual machine and provides a shared execution environment to install, update, run, stop and uninstall bundles without needing to restart the entire system. To minimize the coupling among bundles, the OSGi framework provides a service-oriented architecture that enables bundles to dynamically discover each other for collaboration. An installed bundle can register services by publishing their interfaces using the framework's service registry. This registration makes the services discoverable through the registry so that other bundles can use them. Thus, when a bundle queries the registry, it obtains references to actual service objects registered under the desired service interface. It allows us to search for a certain service when needed, for instance, to search for a pervasive service to execute a behaviour pattern task.

The framework also manages dependencies among services to facilitate coordination among them. These dependencies are implemented by using Wire objects. A Wire object acts like a communication channel between a Producer service and a Consumer service. When a wire is created, the producer service can produce information to be used by the Consumer service. For instance, a wire is created between the context monitor and MAtE so that the context monitor can inform MAtE about context changes. To enable this communication, the Producer service must implement the OSGi Producer interface, while the Consumer service must implement the OSGi Consumer interface. There are two ways to establish communication using a wire: 1) the Producer service can send information to the Consumer service or 2) the Consumer service can request the Producer service for information.

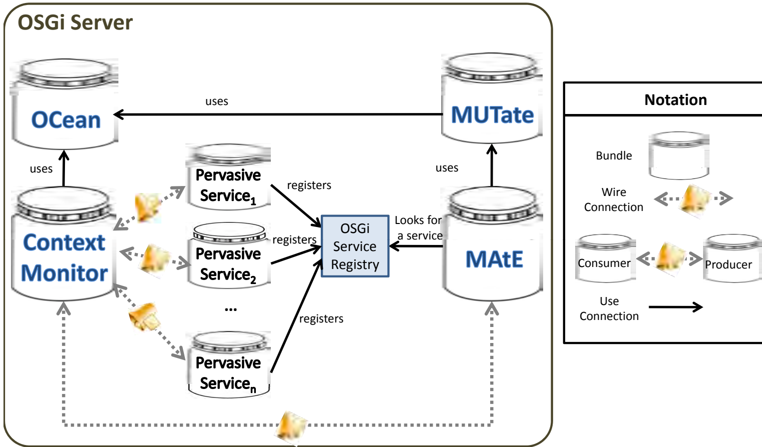

Figure 6.3: Communication among the components of the software infrastructure

In our approach, the communication between services using a wire is always produced from the producer to the consumer.

Figure 6.3 shows how the components of the software infrastructure are connected. As the figure shows, MAtE uses MUTate, in order to interpret the task model at runtime, and the context monitor uses OCean, in order to read and update the context model at runtime. MUTate also uses OCean in order to access to the context model, e.g., for checking context conditions. In addition, MAtE makes use of the pervasive services in order to execute the behaviour pattern tasks. To make this possible, OCean, MUTate and the pervasive services make their interfaces discoverable by publishing them using the service registry.

Furthermore, MAtE and the context monitor are connected by a

wire. In this wire, the context monitor plays the role of producer, because it informs MAtE about context changes, while MAtE plays the role of consumer, because it needs to know the context changes detected by the context monitor. The context monitor is also connected with each one of the pervasive services by a wire. In these wires, the pervasive services act as producers because they provide to the context monitor information about context, while the context monitor acts as a consumer, because it uses the information produced by the pervasive services.

It is worth noting that the relationship of use (use connection in Figure 6.3), is required when a service needs another service by demand; while the relationship of dependency created by using a wire (wire connection in Figure 6.3) is required if the communication has to be established as soon as the producer has the information needed by the consumer.

Next we explain how the components involved in the automation process have been implemented in Java/OSGi technology. For more implementation details, see Appendix A.

Pervasive Services

Our approach attempts to be as independent as possible from the pervasive service implementation. However, so that these services can be used for the current implementation of our approach, they must fulfil the following requirements:

- They must be implemented using the OSGi/Java technology; specifically, each service has to be provided as an OSGi bundle.
- Each service has to be registered as a service in the OSGi service registry by using a unique service.pid. This registry actually stores the interface that the service provides, which allows us to search for a certain method of the service to be executed.
- Each service has to implement the OSGi Producer interface to be prepared for informing the context monitor when context changes

are produced.

- Each service must implement operations for: setting which context properties the service manages; obtaining the values of these context properties; checking if the values of these properties have changed; and notifying its consumers when a context change is produced.

  To facilitate the implementation of these services, we provide the *Service* class that implements all the needed methods except the operation for setting which context properties the service manages, which has to be implemented by each service. Detail information about the implemented *Service* class can be found in Appendix A.

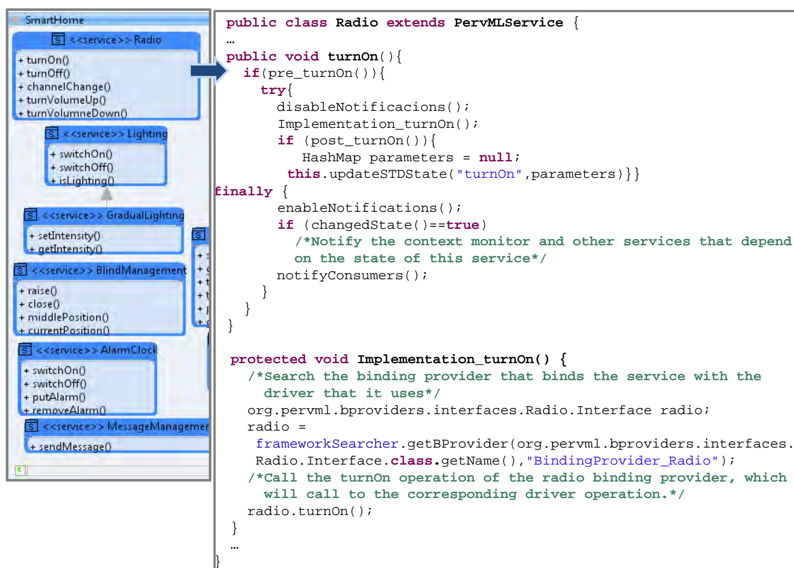

Figure 6.4: Part of a PervML service model and an example of service code generation

To implement the pervasive services used in the case studies developed to test our approach (see Chapter 8), we have used a model-driven development (MDD) method named PervML. This method was presented in (Muñoz *et al.*, April 2006; Serral *et al.*, 2010). PervML allows us to automatically generate Java/OSGi pervasive services that provide the functionality above explained. This generation is achieved by specifying the needed services using a set of high level abstraction models.

For instance, Figure 6.4 shows an example of the PervML service model. In this model, the services, their methods and their relationships are specified. The model shows a service named *Radio* that provides the operations: *turnOn, turnOff, changeChannel, turnVolumeUp*, and *turnVolumeDown.* The figure also shows the implementation of the turnOn service operation automatically generated from the PervML models.

Ontology-based Context model management mechanisms (OCean)

OCean must allow the interaction with the context model for both interpreting it and modifying it. This interaction must respect the vocabulary established in the context ontology to allow the communication using the same high-level abstraction concepts, i.e., at modelling level. For these reasons, we have implemented OCean as a Java API based on the ontology concepts.

To implement this API, we first investigated if there were tools that could help us to develop a Java API for managing an ontology model. Nowadays, there are several free tools that automatically generate a Java API from a given ontology for the handling of OWL instance data. Examples of these tools are: Jastor[1], Jaob [2], Protégé[3] or OWL2Java[4].

After studying and trying these tools, we use the Jastor tool because

[1] http://jastor.sourceforge.net/
[2] http://wiki.yoshtec.com/jaob
[3] http://protege.stanford.edu/overview/protege-owl.html
[4] http://www.incunabulum.de/projects/it/owl2java

it was the only one that generates the methods that we need, avoiding as much as possible concepts dependent of OWL technology in the API. This facilitates the evolution of our approach to new model technologies.

Finally, the OCean API provides a *Factory* class for creating new individuals in the context model and getting those that have already been created. Also, the API provides an implementation class (and its corresponding Java interface) for each one of the OWL classes defined in the context ontology. Each class allows its individual to be created, obtained, modified, and deleted. Thus, the instances of the context model can be managed by using the same concepts defined in the context ontology.

In addition, we have extended OCean with a *Model* class that allows a context model to be opened and saved. To make easier the update of the context model according to context changes, this class also provides a more generic API that allows us to manage the individuals of the ontology independently of its class. Specifically, this *Model* class provides methods such as *setProperty* or *getProperty* to update and obtain a property of any individual; or *addRelatedInstance*, to relate an individual to another individual.

Furthermore, this *Model* class provides facilities for querying the model using SPARQL (see Section 2.3), which is a graph-matching query language recommended by the W3C that allows queries to be built to search for certain individuals in the context model. Specifically, the class provides the method *checkCondition* that receive a SPARQL query in String format and is in charge of checking whether the query is fulfilled or not.

To implement this class, we have used Jena 2.4[5], the OWL API 2.1.1[6], and the Pellet reasoner 1.5.2. (see Section 2.3). Jena is a Java framework for building Semantic Web applications that provides a programmatic environment for OWL and SPARQL and includes a rule-based inference engine. We have used Jena to open the OWL model and save the performed changes in it. The OWL API is an open-source

[5]http://jena.sourceforge.net/
[6]http://owlapi.sourceforge.net/

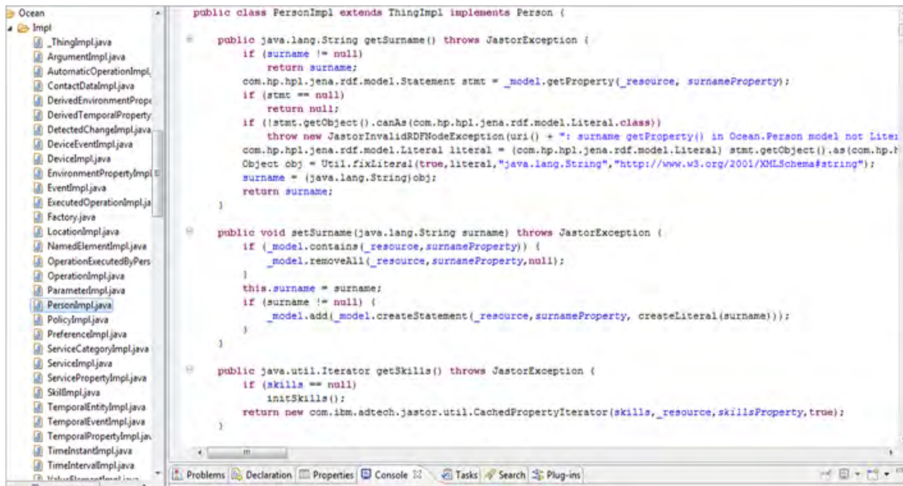

Figure 6.5: Overview of the OCean API

API that provides facilities for creating, examining and modifying an OWL ontology. We have used the OWL API to access to and modify the individuals of the context model. Pellet is an open-source tool that provides reasoning services for OWL ontologies. Pellet facilitates accessing to the information stored in the ontology and allows us to launch a SPARQL query against the context model using Jena.

Figure 6.5 shows an overview of the classes provided by OCean and a partial view of the source code of its *Person* class. Specifically, it shows the methods *getSurname*, to obtain the surname of the person, *setSurname*, to modify the surname of the person, and *getSkills*, to obtain the skills of the person.

Model-Based User Task management mechanisms (MUTate)

Similar to OCean, MUTate must allow the interaction with the task model for both interpreting it and modifying it. This interaction must respect the vocabulary established in the task model metamodel to allow the communication using the same high-level abstraction concepts,

i.e., at modelling level. For these reasons, we have also implemented MUTate as a Java API based on the concepts defined in the task model metamodel.

Since the graphical tool for specifying the task model has been developed using Eclipse (Eclipse, 2011), we have also used the modelling plugins that it provides to implement MUTate.

In particular, we have used the EMF and EMF Model Query (EMFMQ) plugins of the Eclipse Platform, which provide us with many benefits for managing an XMI model at runtime.

From the metamodel of the task model in Ecore (see Figure A.3), we use EMF to generate a Java API for managing a task model. The generated API provides a Factory class for creating new instances of the task model metamodel elements and getting those that have been already created. In addition, MUTate provides a Java interface and an implementation class for managing the instances of each one of the classes of the metamodel. These generated classes provide get and set methods to access and change the information of the instances specified in the model.

Some of these Java classes represent context conditions, such as the classes *ContextSituation* or *ContextPrecondition*. We have also implemented in these classes the *checkCondition* method to check whether the condition is fulfilled or not. This method interprets the logical expression of the condition and builds a query in SPARQL. Once the query has been built, the method uses the *Model* class of OCean to launch it against the context model.

EMFMQ facilitates the process of search and retrieval of model elements in a flexible, controlled and structured manner. To achieve this, this plugin allows the construction and execution of queries in a SQL-fashion. We use these queries to search for and get the instances of the model that need to be accessed or modified.

Figure 6.6 shows an overview of the classes provided by MUTate and a partial view of the source code of its implementation classes *TaskModel* and *BehaviourPattern*. Specifically, the figure shows the *getBehaviourPatternByContextSituation* method of the *TaskModel*

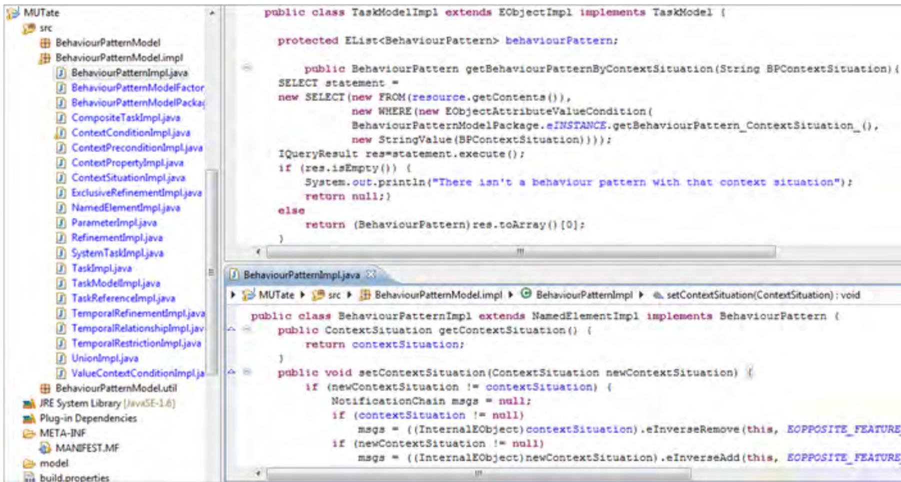

Figure 6.6: Overview of the MUTate API

class and the *getContextSituation* and *setContextSituation* of the *BehaviourPattern* class. The *getBehaviourPatternByContextSituation* method returns the behaviour pattern whose context situation is the same than the *BPContextSituation* argument value. To find the corresponding pattern, it searches for it by using a query statement built with EMFMQ. The *getContextSituation* method obtains the context situation of the behaviour pattern, while the *setContextSituation* method modifies it.

The Context Monitor

The context monitor is in charge of updating the context model according to the context changes. These context changes are physically detected by the pervasive services (above explained) that control the system devices. Thus, in order to capture context changes, the monitor is continuously monitoring the execution of the pervasive services.

To do this, the context monitor implements the Consumer interface and creates a wire with each service.

Thus, when a change in a service is produced, the service notifies the context monitor about this change, since the monitor is a consumer of the service. In this notification, the service sends to the context monitor a hashmap that contains the context variables whose value has changed. It can be seen like if this data was sent through the wire from the consumer to the producer.

When a change notification is produced, the updateContextModel method is called in order to reflect the changes in the context model. In order to update the context model, OCean is used. In a change notification, the context monitor receives the name of each changed individual, the changed properties and their new value. Using the OCean Model class, the monitor changes the values of the corresponding properties.

Finally, once updated the context model, the updated method of the context monitor must inform MAtE about the context that has been updated. To do this, the context monitor and MAtE are also related by a wire. In this case, the context monitor plays the role of producer (implementing also the Producer interface), while MAtE plays the role of consumer (implementing the Consumer interface).

Thus, when the context monitor updates the context model, it notifies MAtE about the corresponding context change by calling the notifyConsummers method implemented in the context monitor. In this notification, the context monitor sends to MAtE a hashmap that contains the context properties whose value has changed.

MAtE

To automate the behaviour patterns in the opportune context, MAtE must check if any behaviour pattern has to be executed when a change in context is produced. For this reason, the context monitor notifies MAtE about any context change. To allow this, MAtE implements the Consumer interface and creates a wire with the context monitor.

When MAtE receives the notification, it carries out the following steps (which are summarized in Figure 6.7):

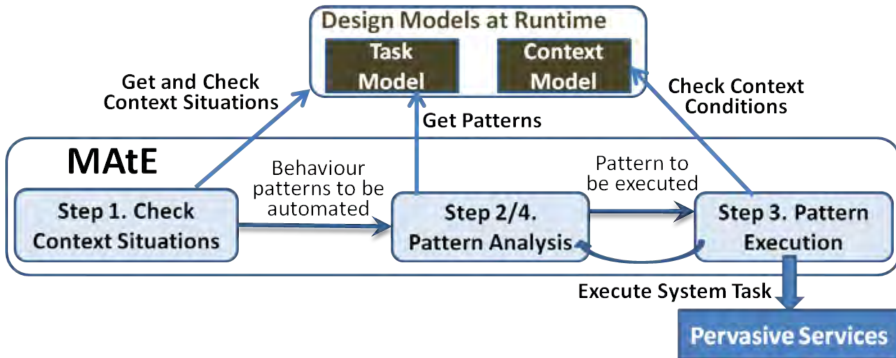

Figure 6.7: MAtE process for automating the user behaviour patterns

1. Check the fulfilment of the context situations specified in the task model. To do this, MAtE first obtains the behaviour patterns specified in the task model by using the *getBehaviourPattern* method of the *TaskModel* class of MUTate. MAtE then analyses the context situation of each behaviour pattern to identify which ones depend on the notified context change. If a context situation depends on the notified context change, MAtE queries the context model to check whether it is fulfilled by using the *checkCondition* method of the *ContextSituation* class. Finally, when a context situation is fulfilled, MAtE adds its behaviour pattern to a list that stores the behaviour patterns that have to be executed.

2. If there is some behaviour pattern to be executed in the list, MAtE analyses their priorities and starts to execute the behaviour pattern with the highest priority. If there are several patterns with the same priority, this means that their execution order does not mind, therefore, the first one of the list is the first one that is executed.

3. MAtE executes the behaviour pattern according to its refinements, the temporal relationships specified among its tasks and the up-to-date context information (stored in the context model) on which tasks and relationships depend:

3.1. Its first refinement is obtained by using the *getRefinement* method (implemented in the *CompositeTask* class and inherited by the *BehaviourPattern* class).

3.1.1. If it is an exclusive refinement, MAtE first gets all the subtasks. Following the order of the refinements, MAtE searches for the first subtask that can be executed and executes it. If the task is a composite task, MAtE executes its *executeCompositeTask* method, which goes to the step 3.1 following a recursive process. If the task is a system task, MAtE executes its method *executeSystemTask*. This method first searches for the pervasive service related to the task by using the OSGI capabilities, which allow services to be searched at runtime. Then, MAtE executes the service by using the Java reflection capacities, which allow us to execute a method by using its name, its arguments and its class name (i.e., the service name). The process followed by this step is summarized in the Algorithm 1.

3.1.2. If it is a temporal refinement, all the subtasks of the composite task must be executed in the appropriate order. MAtE gets all the subtask and performs them following an iterative process and according to the order established by the temporal relationships. Thus, MAtE starts by carrying out the first subtask. If the subtask has a context precondition, MAtE only executes it if this precondition is satisfied. If the task is a composite task, MAtE executes it by using its executeCompositeTask method, which goes to the step 3.1 following a recursive process. If the task is a system task, MAtE executes it by using its method executeSystemTask. Once a task is executed, MAtE checks the type of the temporal relationship that associates it with the next task. If the temporal relationship is >>[c]>> (*enabling when C is satisfied*) MAtE waits until the condition c is satisfied to carry out the next subtask; in the same manner, if the

relationship is t>> (*enabling after T minutes*) MAtE waits t minutes. However, if the relationships are *Task-Independence* or *Enabling* MAtE does not have to wait and the next task is directly carried out. The process followed by this step is summarized in the Algorithm 2.

4. If there are more behaviour patterns in the list of behaviour patterns for being executed, MAtE gets the one with the next highest priority. If its context situation is still satisfied, MAtE goes to step 3 in order to execute the behaviour pattern. If the situation is not satisfied, MAtE rules out the pattern and executes again the step 4. This step is performed until all the behaviour patterns obtained in the step 1 have been analysed.

Algorithm 1 Exclusive Composite Task Execution.

```
subtaskList=getSubtasks(ct)
for all subtask in subtaskList do
  if subtask.precondition satisfied then
    executeTask(subtask)
    return
  end if
end for
```

As an example, we next explain the process that MAtE follows to execute the WakingUp behaviour pattern (see Figure 5.4). Every minute, the context monitor updates the *CurrentTime* context property and notifies MAtE about this change. Since the context situation of the WakingUp pattern depends on this property, MAtE checks whether its context situation is then satisfied.

When it is 7:50 a.m. and a working day, the context situation of the WakingUp behaviour pattern is satisfied and MAtE has to carry out the pattern. This behaviour pattern has 4 subtasks refined by temporal refinements. This means that these tasks must be executed following the order established by the temporal relationships between the tasks. Thus, MAtE starts to perform them by getting the subtask refined

Algorithm 2 Temporal Composite Task Execution.

```
subtaskList=getSubtasks(ct)
for all subtask in subtaskList do
  if subtask.precondition satisfied then
    executeTask(subtask)
  end if
  TempRel=subtask.getTemporalRelationship()
  if TempRel is ENABLING_AFTER_T_MINUTES then
    Wait T minutes
  else if TempRel is ENABLING_WHEN_C_IS_SATISFIED then
    Wait until C is satisfied or the temporal restriction goes off
  end if
end for
return
```

by the first refinement. This subtask is the *turn on bathroom heating* task. It has a context precondition (*BathroomTemperature<28*) which is first checked by MAtE using the checkCondition method. If this method returns true, the task is executed. This task is a system task and is related to the *BathroomHeating* service and its *switchOnHeating* method whose execution carries out the task. Thus, MAtE executes its method executeSystemTask to execute the *switchOnHeating* method of the *BathroomHeating* service.

The *turn on bathroom heating* task is related to the next task by the 10 min>> relationship. This means that MAtE must wait 10 minutes before executing the next task. This task is the *turn on the radio*, which is also a system task and is related to the *Radio* service, its *turnOnRadio* method, and the *favouriteRadioChannel* context parameter. Thus, ten minutes later, MAtE executes the task by calling the executeSystemTask. It searches for the *Radio* service and executes its *turnOnRadio* method with the value of the *favouriteRadioChannel* parameter previously obtained from the context model.

The executed task is related to the next task by the | = |

relationship. This means that MAtE does not have to wait to execute the next task. This task is the *lighting* composite task, which is refined by exclusive refinements. This means that only the first task that can be executed, must be executed. Thus, MAtE gets the subtask of the first refinement, which is the *raise the bedroom blinds* and checks its context precondition (*sunnyDay=true*). If it is satisfied, MAtE executes this subtask by using the executeSystemTask method; otherwise, MAtE gets the subtask of the next refinement, which is the *switch bedroom light off* task. If its context precondition (*sunnyDay=false*) is satisfied, MAtE performs this task by using the executeSystemTask method.

Then, MAtE passes to carry out the last task. This is related with the previous task by the >>[BobLocation=Kitchen]>>, which means that the last task must be executed when Bob enters in the kitchen. This is checked by using the checkCondition method. Finally, when this condition is satisfied, MAtE calls its executeSystemTask method for executing the *makeCoffee* method of the *CoffeeMaker* service.

6.4 Conclusions

In this chapter, we have explained how the user behaviour patterns specified in the context and task models are automated at runtime in the opportune context.

Specifically, we have first described the requirements that have to be taken into account to perform this automation. Next, we have explained the automation process. This process defines the steps that are followed in order to automate the behaviour patterns by fulfilling the detected requirements.

In addition, we have described the software infrastructure that achieves the automation of the user behaviour patterns specified in the models by carrying out the automation process. This infrastructure is composed of 1) the set of pervasive services provided for controlling the devices of the smart environment; 2) OCean and MUTate, the high-level abstraction mechanisms for managing the context model and the task model, respectively; 3) the context monitor that monitors the context

changes and updates the context model accordingly, and 4) MAtE, the automation engine capable of automating the behaviour patterns as specified in the models. Using these components, behaviour patterns can be automated by only specifying them using the proposed models.

Chapter 7

Addressing the Evolution of the User Behaviour Patterns

Some of the behaviour patterns specified to be automated by the system might never change in user lives; however, most of them will. Users' context and circumstances usually change over time and the automated behaviour patterns must evolve to adapt to these changes. Otherwise, the automation of these behaviour patterns not only may become useless for end-users but may also become a burden on them instead of being a help in their daily life. Although the proposed models specify the patterns to be automated in such a way that their execution adapts to context, changes in user behaviour patterns cannot be anticipated at design time. This makes that the evolution of the automated behaviour patterns is a need to properly automate them.

In this chapter, we explain the mechanisms and tools that are defined and implemented to address this evolution. First of all, it is important to note that the model interpretation strategy used in our approach facilitates to perform this evolution by directly changing the models (i.e., at modelling level), which is one of the top challenges in software

evolution research (Ajila & Alam, 2009; Bennett & Rajlich, 2000; Mens, 2009). Using this strategy, the models are directly interpreted at runtime by MAtE in order to execute the specified behaviour patterns in the opportune context. Thus, as soon as the models are changed to adapt the patterns, the changes are also taken into account by MAtE. This strategy gives us three immediate benefits to perform the evolution of the behaviour patterns:

1. We do not need to maintain the consistency between the modelling of the behaviour patterns and their implementation when modifications are applied.

2. The evolution can be managed more intuitively using concepts of a high level of abstraction (i.e., modelling language).

3. The models can provide us with a richer semantic base for runtime decision-making related to system adaptation.

From these premises, we address the behaviour pattern evolution by adapting their specification in the context and task models.

In order to adapt these models at runtime, the OCean and MUTate APIs explained in Section 6.3 can be directly used. However, although these mechanisms use the same high-level concepts created in the modelling language, only computer technicians could use them. Thus, since the needed changes can be only detected after system deployment, we also design and develop a graphical tool that provides end-users with intuitive interfaces to facilitate the evolution of the automated behaviour patterns at runtime and at a high level of abstraction.

Thus, the rest of the chapter is organized as follows. First, Section 7.1 precisely characterizes the evolution confronted in this work by following the taxonomies published in (Buckley *et al.*, 2003; Lientz & Swanson, 1980). Next, Section 7.2 explains how OCean and MUTate can be used for supporting the evolution of the specified behaviour patterns. Section 7.3 describes the provided tool. Finally, Section 7.4 concludes the chapter.

7.1 Evolution Characterization

Lientz and Swanson (Lientz & Swanson, 1980) classify the software change by answering to the question *why*. They describe three intentions for software change: to perfect the system (perfective), to adapt the system (adaptive) and to correct the system (corrective). According to their descriptions, our evolution is considered as adaptive because user behaviour patterns need to be evolved to adapt to new users' needs and circumstances.

Buckley et al. (Buckley *et al.*, 2003) complete the taxonomy of Lientz and Swanson to characterize the software change answering to the questions: *when, where, what*, and *how*.

The *when* dimension characterizes evolution from two main aspects: (1) the phase of the software life-cycle on which it is performed, which delimits three types of evolution: at compile-time (static), at load time, and at run-time (dynamic); and (2) the anticipation of the required evolution, which delimits two types of evolution: anticipated, if evolution can be foresee at design time; and unanticipated, if evolution needs arise from using the system.

According to this dimension, the confronted evolution is dynamic and unanticipated. It is dynamic because the changes are made on demand at runtime. Thus, we achieve that the costs and risks associated with shutting down and restarting the system for an update are mitigated. It is unanticipated because the changes arise over time after the system deployment; therefore, they cannot be foreseen during the design phase.

The *where* dimension characterizes the software artefacts where changes are made (requirements, architecture, design, source code, documentation or test suites). This dimension takes into account also the following factors: granularity, which refers to the scale of the artefacts to be changed and can range from very coarse to a very fine; the impact of the change and the change propagation, which indicate the process of synchronizing other artefacts that also depend on the change.

In the confronted evolution, the artefacts that are changed are the context model and the task model, and the changes that can be performed in these models are of fine grained because any element of the models can be changed. In addition, since the behaviour patterns are automated by directly interpreting the models at runtime, the changes in the models are automatically propagated in the system, having an impact over all the system.

The *what* dimension characterizes evolution from the system attributes that may condition it. These attributes are: availability (whether the software system has to be permanently available or not); activeness (evolution is either reactive, if system changes must be driven by an external agent, or proactive, if the system is able to self-change by using the information received from monitors); openness (whether or not the system must allow extensions); and safety (whether or not safety aspects must be considered at compile time and/or at runtime). The attributes of this dimension of the confronted evolution are:

- The system has to be always available since it may be needed at any time.
- The system is reactive to the evolution because it has to be driven externally; otherwise, the evolution of the behaviour patterns may bother users and make them lose system acceptance.
- The system is open because it is specifically built to allow its evolution.
- The system must provide behavioural safety, which means that the evolution mechanisms must prevent or restrict undesired behaviour changes at runtime. A certain degree of static safety is achieved at compiling time because the changes must be in accordance to the task metamodel and the context ontology. For dynamic changes, tests must be performed to ensure behavioural safety.

The *how* dimension characterizes evolution from the degree of automation and formalism that is introduced in the mechanisms

provided to support it. The confronted evolution is supported with a high degree of automation and formality because the behaviour patterns are evolved by updating the task and context models, which are of a high level of abstraction.

Regarding the question *who*, it has not a taxonomy because it is specific for each software change (Buckley *et al.*, 2003). In our case, the evolution of the automated behaviour patterns can be driven by the system designers; however, it is convenient for end-users to be able to drive the evolution since it is necessary after system deployment.

7.2 Mechanisms for Evolving the Behaviour Patterns

As explained, as soon as the models are changed to adapt the patterns, the changes are applied into the system because MAtE directly interprets the models to execute the behaviour patterns. Thus, to evolve the behaviour patterns, the mechanisms provided for managing the models at runtime OCean and MUTate (see Section 6.3) can be directly used. These mechanisms use the concepts of the language defined for specifying the models (task, behaviour pattern, user, preference, etc.), which facilitates the understanding, handling and evolution of the automated behaviour patterns. In addition, these mechanisms ensure that the changes are in accordance with their metamodel definition and, therefore, syntactically correct. In this way, these mechanisms define how the patterns can be changed over time and also maintain software quality characteristics.

It is worth noting that the use of such mechanisms raise the evolution level to the modelling level, which allows the system to be evolved by using high level abstraction concepts instead of by changing lines of code. Furthermore, the mechanisms are implemented in Java and are provided as APIs; therefore, they can be imported and used by any Java application.

Thus, using these high-level mechanisms, any change that respects the task metamodel and context ontology syntaxes can be performed

to evolve the automated user behaviour patterns. These mechanisms allow new behaviour patterns to be created, and also to enable/disable, modify or delete those that are already specified. Next, we show the types of evolution that may be needed to modify an automated behaviour pattern. Evolutions for creating new behaviour patterns or deleting them are performed in an analogous way.

Evolving the executed services. New tasks may be required to be automated in a behaviour pattern, other tasks may be not needed anymore, and some tasks may need to be slightly modified. For instance, in the *WakingUp* pattern, the user may want that the system informs them about the weather when s/he is in the kitchen. The user may also want that the lights of the bedroom are not switched on and instead of waking up with the radio he may want to be woken up with relaxing music. To perform this type of evolutions, the tasks of a behaviour pattern can be changed.

Modifying the turn on the radio task

```
String newName="turn on relaxing music";
task_turnOn.setName(newName);
task_turnOn.setID("WakingUp_"+newName.replace(" ", ""));
task_turnOn.setServiceName("Music");
task_turnOn.setServiceMethodName("turnOn");
Argument musicType= taskModelFactory.createArgument();
task_turnOn.setTemporalRelationship(task_lighting.getTemporalRelationship());
```

Removing the lighting task

```
wakingUp.getRefinements().remove(task_lighting.getRefinedFrom());
```

Creating the new task

```
SystemTask newSystemTask= taskModelFactory.createSystemTask();
String nameNewTask="inform about the current weather";
newSystemTask.setName(nameNewTask);
newSystemTask.setID("WakingUp_" + nameNewTask.replace(" ", ""));
newSystemTask.setServiceName("Weather");
newSystemTask.setServiceMethodName("inform");
Refinement refinement= taskModelFactory.createRefinement();
refinement.setType(TaskRefinementType.TEMPORAL);
refinement.setRefinementFrom(wakingUp);
refinement.setRefinementTo(newSystemTask);
TemporalRelationship tempRel=taskModelFactory.createTemporalRelationship();
tempRel.setType(TemporalRelationshipType.ENABLING);
tempRel.setTemporalRelationshipFrom(task_makeCoffee);
tempRel.setTemporalRelationshipTo(newSystemTask);
```

Figure 7.1: Evolving the executed services using OCean and MUTate

Figure 7.2: WakingUp model before and after evolving the executed services

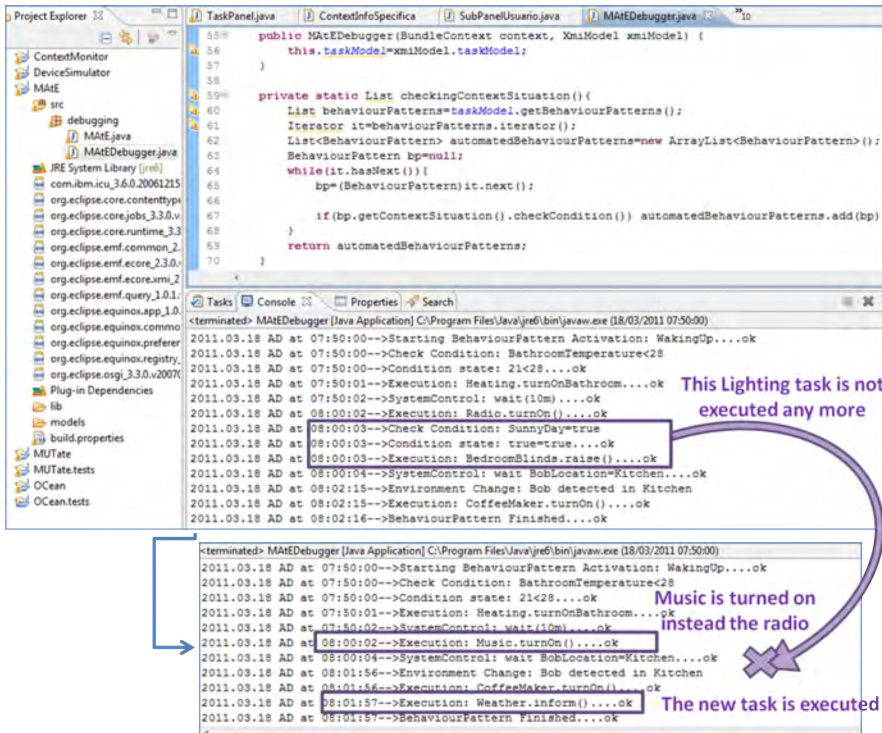

Figure 7.3: Execution traces before and after evolving the executed services

Figure 7.1 shows how to modify the services executed in the *WakingUp* behaviour pattern using OCean and MUTate. As shown, the turn on the radio task has been modified to turn on relaxing music. The lighting task and its subtasks have been removed; therefore, it will not be executed anymore in the pattern. And a new task named inform about the current weather has been created in the pattern and has been related to the make coffee task using a temporal relationship of the enabling type. Figure 7.2 shows how the *WakingUp* pattern is modified by executing this code, while Figure 7.3 shows how the execution of the pattern changes. These traces have been obtained by executing the system in a debug manner and simulating the fulfilling of the context situation of the *WakingUp* pattern. As we can see, the pattern is executed according to the performed evolutions.

Modifying context conditions. The conditions where the services must be executed may change over time. For instance, the user may want to be woken up a half an hour later (his timetable may change) and only if he is not sick. He may also want the heating to be turned on in the bathroom for 15 minutes instead of 10 (so that the bathroom is warmer when he takes a shower). To perform this type of evolution, user information can be added in the context model and the context conditions used in the specification of a behaviour pattern can be changed. Figure 7.4 shows how to change the context conditions of the Waking Up behaviour pattern using OCean and MUTate. As shown, first of all, a new user property is added to indicate whether the user is sick or not. Then, the context situation of the pattern has been changed to make it dependent on the created context property. A new condition has been added to the context situation so that the pattern only is enabled if the user is not sick. Also, the time in which the pattern has to be enabled has been changed to 8:15. Thus, the *WakingUp* pattern will be enabled at 8:15, in working days and only when the user is not sick. Also, the temporal restriction of the relationship between the two first tasks of the pattern has been changed to 15 min. Figure 7.5 shows the *WakingUp* pattern and how this pattern is executed after performing these evolutions. As we can see in the traces, the behaviour pattern is executed according to the evolutions.

Adding a new property to the user to indicate whether he is sick

```
//Create the new user property
UserProperty newUserProperty=contextModelFactory.
                              createUserProperty("Bob_Sick",contextModel);
newUserProperty.setName("Sick");
newUserProperty.setValue("false");

//Relate the preference with the user
Person bob=contextFactory.getPerson("Bob", contextModel);
newUserProperty.setOfPerson(bob);
```

Modifying the context situation

```
//Add the new condition "sick=false"
Union and=BehaviourPatternModelFactory.eINSTANCE.createUnion();
and.setUnionOperator(UnionOperator.AND);
ContextCondition contextCondition=taskModelFactory.createContextCondition();
ContextProperty sick=taskModelFactory.createContextProperty();
sick.setInstanceName("Bob_Sick");
sick.setPropertyName("value");
contextCondition.setLeftPart(sick);
contextCondition.setOperator(ComparativeOperator.EQUAL);
contextCondition.setRightPart("false");
and.setContextCondition(contextCondition);
ContextSituation contextSituation=wakingUp.getContextSituation();
contextSituation.getUnion().getContextCondition().setUnion(and);

//Changing the condition "CurrentTime=7:50" por "CurrentTime=8:15"
contextSituation.getRightPart().setValue("08:15");
```

Modifying the relationship that relates the turn on bathroom heating task to the next task

```
task_turnOn.getTemporalRelationship().getTemporalRestriction().setMinutes(15);
```

Figure 7.4: Modifying context conditions using OCean and MUTate

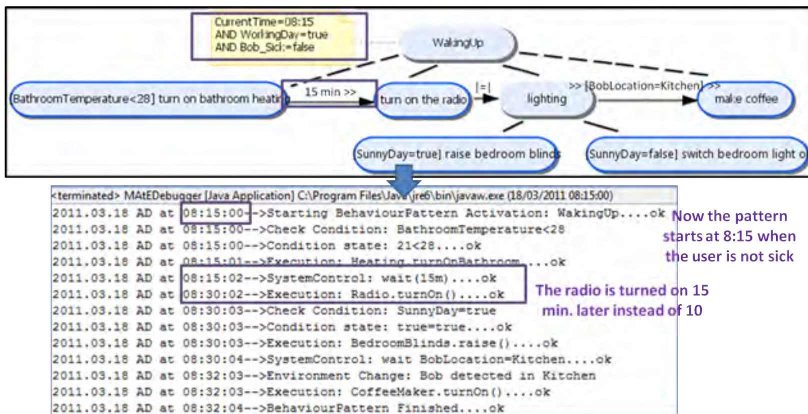

Figure 7.5: WakingUp model and execution trace after modifying the context conditions

Modifying the execution order of the services. User may want that the services of a behaviour pattern are carried out in another order. For instance, the user may want to have breakfast before taking a shower and that the coffee is made just before the radio is turned on, thus, the coffee would not be so hot when he takes it. To perform this type of evolution, the relationships between the tasks of a behaviour pattern can be changed. Thus, the first task to be automated should be the make coffee task, then the user must be woken up and after this, the bathroom heating must be switched on (the time for having breakfast is enough so that the bathroom is warm).

Figure 7.6 shows how to change the execution sequence of the services executed in the *WakingUp* behaviour pattern using OCean and MUTate. As shown, the first task is moved to be executed the last task and the last task is moved to be executed the first task. The relationships of these tasks have been also changed. In addition, the context situation of the pattern has been also changed because the pattern must start when the user must be woken up. Figure 7.7 shows the *WakingUp* pattern and how this pattern is executed after performing these evolutions. As we can see in the traces, the behaviour pattern is executed according to the evolutions.

Moving the bathroom heating task to the last task to be automated in the pattern

```
task_bathroomheating.setTemporalRelationship(null);
task_lighting.getTemporalRelationship().setType(TemporalRelationshipType.ENABLING);
task_lighting.getTemporalRelationship().setTemporalRelationshipTo(task_bathroomheating);
```

Moving the make coffee task to the first task to be automated in the pattern

```
TemporalRelationship tempRel = taskModelFactory.createTemporalRelationship();
tempRel.setType(TemporalRelationshipType.ENABLING);
tempRel.setTemporalRelationshipFrom(task_makeCoffee);
tempRel.setTemporalRelationshipTo(task_turnOn);
```

Updating the context situation

```
ContextSituation ContextSituation=wakingUp.getContextSituation();
ContextSituation.getRightPart().setValue("08:00");
```

Figure 7.6: Evolving the service execution plan using OCean and MUTate

Thus, MUTate and OCean allow the behaviour patterns to be evolved at runtime by using concepts of a high level of abstraction

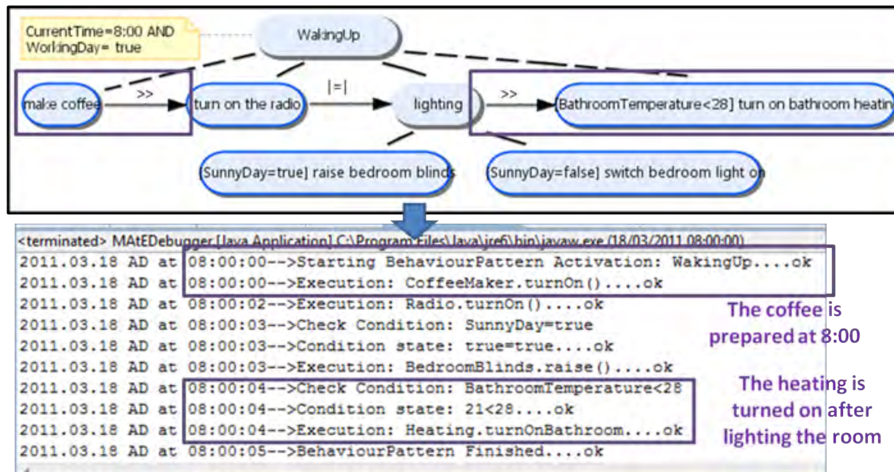

Figure 7.7: WakingUp model and execution trace after evolving the service execution plan

(Preference, ContextSituation, BehaviourPattern, etc.), which are easy for developers to understand and use.

7.3 Tool Support

The above presented mechanisms can be used by any Java application to adapt the automated behaviour patterns to user needs after system deployment. In this section, we use these mechanisms to develop a graphical tool that facilitates this adaptation. This tool provides user-friendly interfaces that update the models according to the performed changes by using MUTate and OCean. Thus, the tool can be also used to evolve the behaviour patterns at runtime without the need to stop or redeploy the system.

This tool provides users with the following functionalities:

- *Context Management:* the tool shows users the context information for which they have permission according to their permission policy (which is specified for each user in the context

model; see Section 5.1.3). Also, using this functionality, users can modify this information or delete it if it is not used in the task model. Thus, end-users could manage their information (e.g., their preferences, contact information, etc.) and see the system information; while an administrator could have permission for managing all the context information (e.g., managing users, services, environment properties, etc.).

- *Behaviour Pattern Management:* the tool allows users to add, modify, or delete behaviour patterns by facilitating the information necessary to do this. In addition, if users do not want certain patterns to be executed during a period of time, our tool also allows them to configure these patterns to be enabled or disabled.

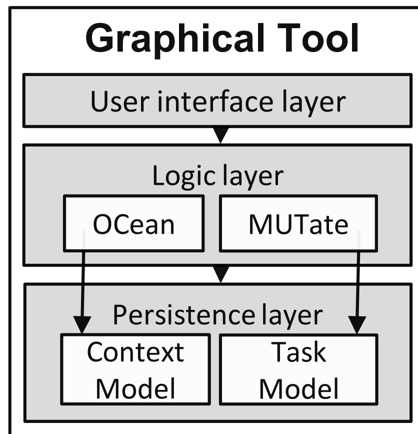

Figure 7.8: End-user toolkit architecture

To provide these functionalities, the toolkit is organized in layers with well defined responsibilities following the Layers Architectural Pattern (Gamma *et al.*, 1994). Thus, the toolkit has been designed in the following three layers, which are shown in Figure 7.8:

- **The User Interface Layer**: This layer provides users with

graphical user-friendly interfaces to adapt the behaviour patterns.

- **The Logic Layer**: This layer is composed by OCean and MUTate (see Chapter 6), mechanisms that allows the interface layer to retrieve and update the context information of the context model and the behaviour pattern information of the task model at runtime.
- **The Persistence Layer**: This layer provides persistence to the managed information. It is composed of the models explained in Chapter 5: the context model, where the context information is stored in OWL (Ontology Web Language), and the context-adaptive task model, where the behaviour pattern information is stored in XMI (XML Metadata Interchange).

Since the Logic and Persistence Layers have been already explained in previous chapters, in this section we focus on describing the User Interface Layer in detail.

The User Interface layer is responsible for the interaction with users. It provides a set of user graphical interfaces that have been developed using the Natural Programming design process (Myers *et al.*, 2004). This process applies the principles of user-centred design for the purpose of treating usability as a first-class design objective. This avoids subordinating usability in favour of historical convention, designer preference, or theoretical elegance. The steps of the Natural Programming design process are as follows: step 1) identify the target audience; step 2) understand the target audience's language, techniques, and thinking for problem solving; step 3) design the new tool; and step 4) evaluate it.

Following these steps, we first identified the target audience of the tool as the end-user of the system. In this thesis, we focus on end-users with computer knowledge (at least knowing how to use a computer and common programs such as Microsoft Office Word or Excel[1]).

Second, we determined the language that these users commonly use to describe a behaviour pattern. To do this, we first selected a

[1] http://www.microsoft.com/latam/office/

representative group of end-users, who also participated for developing the case studies that will be introduced in Chapter 8. They were a total of 18 people with different intellectual capacities and studies, covering a wide variety of professions, ages and computers' knowledge.

To determine how the target audience describe behaviour patterns, we first asked these end-users to define a task. As stated in the literature, this term was perfectly understood by them. They defined a task as work to be done, like it is described in the common definition of task. Then, we asked them to determine how they refer to a set of tasks habitually performed. Most of them used the term *routine* and described the context situation that triggers the routine by answering to the question *when* or *in which circumstances*. Finally, we asked them to define or put an example of the rest of concepts that we usually use, such as *behaviour pattern*, *condition* and *context*. Generally, participants understood a behaviour pattern as something that is composed by several routines, like how a patient behaves. The concepts *condition* and *context* were not familiar by everybody; however, all the users easily understand them after explaining the meaning of these concepts.

In addition, we also observed the information of the scenarios collected for performing the case studies (see Chapter 8) in which users described tasks that they performed daily. To organize the tasks participants used the words *when, if, after, while, before* and *at a certain time*.

Once known the language used by the end-users to describe behaviour patterns, we develop the interfaces of the tool according to this language. To develop these interfaces we follow Visual Programming approaches (Mellon, 2009; Pérez & Valderas, 2009) and good-practices in End-user Development (Galitz, 2002; Lieberman *et al.*, 2006; Nielsen, 1993; Welie & Traetteberg, 2000). Next, we first explain the main design interface decisions that we have applied, then we describe the developed interfaces, and finally we explain how to use them to evolve the behaviour patterns.

Regarding the evaluation of the tool, which is the last step of the Natural Programming process, it will be explained in Chapter 8 where

the whole proposed approach is validated.

7.3.1 Interface Design Decisions

Several studies have published very useful and successful advices and design patterns for greatly improving the usability of user interfaces. We have tried to apply them for treating usability as a first-class design objective. Specifically, the design patterns that we have applied in the interfaces for managing the behaviour patterns are the following:

- **Displaying the elements using a grid layout**: to allow user to quickly understand information, it is recommended to arrange all objects in a grid using the minimal number of rows and columns, making the cells as large as possible (Welie & Traetteberg, 2000). Using a grid layout, the objects of the screen are clearly organized and grouped conceptually. This improves the presentation of the information minimizing the time to scan, read and view the objects on the screen.

- **Offering navigation buttons**: to allow users to access an amount of information which cannot be put on the available space, it is recommended to show the information in several spaces and allow the user to navigate between them (Welie & Traetteberg, 2000). Each one of these spaces can group the information into categories that match the user's conceptual model of the data. In addition, navigation buttons suggest end-users that they are navigating a path with steps. This improves the learning and memorization of the task to be performed in each step. These navigation buttons are recommended to be put at the top or left to reduce the needed screen space and make it easier for users to follow the steps.

- **Using a tree representation**: a tree view is a well-known organization scheme to structure the information in a hierarchy of generality or importance. This is a good scheme because people are very familiar with, and have an excellent mental model of

this organization. Such a structure provides information about information sequence, information quantity, and the relationships existing between components. Thus, this scheme is recommended when there is a lot of items to show that can be grouped into a large number of categories. First, the information units are identified and then they are organized in categories according to importance or generality, from general to specific. Also, for not overloading, it is recommended to include buttons to expand or collapse the hierarchy. In addition, when an item is selected, it is also recommended that its parameters and their actual values are displayed (Galitz, 2002; Lieberman *et al.*, 2006).

- **Offering options to select rather than introduce text**: when it is needed that users supply the application with data, it is recommended to use selection field from a set of options instead of an entry field. This ensures that users enter data in the correct syntax and allows information to become less subject to spell or type errors (Galitz, 2002; Welie & Traetteberg, 2000).

7.3.2 Description of the Graphical User Interfaces

Applying the above design decisions, we have developed user-friendly interfaces for providing the Behaviour Pattern Management and Context Management functionalities by using Swing (Loy *et al.*, 2002) and the Eclipse Platform.

As determined in the second step of the natural programming process, we use the term routine in the interfaces provided for managing behaviour patterns. This term is more familiar to end-users than the term behaviour pattern. We also avoid the use of the term *context information* because it is not a term commonly used by end-users. Thus, we separate the context management into:

- User Information Management: the tool shows users their information managed by the system, such as preferences, contact information, etc. The tool also allows users to add new properties corresponding to his/her information, and modify the shown

properties, or delete them if they are not used in the automated behaviour patterns.

- System Information Access: the tool shows users the context information managed by the system and for which they have permission according to their permission policy (see 5.1.3). Specifically, the tool shows users information about the services, the environment and temporal information.

All the interfaces designed to evolve the behaviour patterns share a similar structure and are displayed using a grid layout. At the top of the interface, we guide users by using navigation buttons that indicate the previous, current, and next steps to perform in order to achieve the corresponding goal. The rest of the interface is divided into two columns:

- The left column is divided into two rows. The top and bigger row of the column contains the working area. In this area, users perform the corresponding step. It provides 1) instructions to help users to complete the step; 2) auto completion to reduce errors and user effort; and 3) warning messages to warn users about errors committed when introducing the information (e.g., the user sets a text value rather than a numeric value) and to warn them about necessary information that has not been introduced. The bottom row contains the information area. In this area, explanations about what the user is doing are provided in natural language. This give them feedback and also can help them to see possible committed errors.

- The right column shows users all the information that they need to be able to complete the step. Thus, end-users just need to select the information from the right column and drag it to the proper location in the working area.

With regard to the Behaviour Pattern Management functionality, it is the most important functionality and the most frequently used. For this reason, after logging in the system, the user sees this funcionality

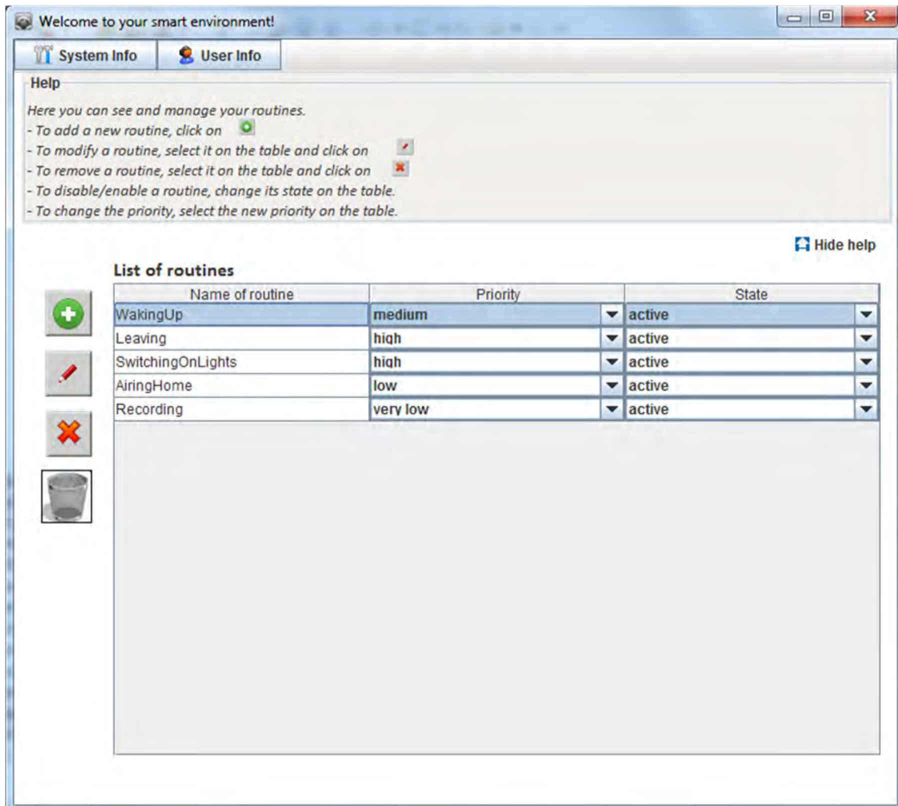

Figure 7.9: Snapshot of the end-user tool for managing the user behaviour patterns

as the initial interface of the tool. This interface shows to the user the behaviour patterns for which s/he has permission, allowing the user to navigate to the other offered functionalities (system information and user information). This initial interface is shown in Figure 7.9. This interface shows the name, the priority and the state of each behaviour pattern, allowing users to change these properties. In addition, the interface also provides users with the operators for modifying and deleting a behaviour pattern, and for creating a new one. If the user selects to modify a pattern, s/he can select to modify the context

situation or the tasks of the pattern. If s/he selects to add a new behaviour pattern, end-users have to specify the context situation and the tasks of the pattern. For each one of these steps, we have developed an interface like the one described above.

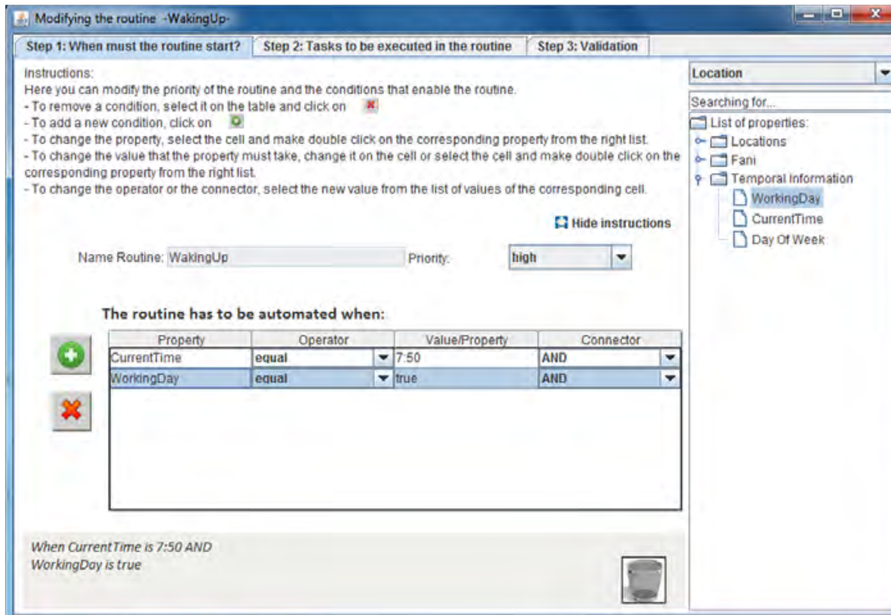

Figure 7.10: Snapshot of the end-user tool for specifying the context situation of a pattern

For instance, Figures 7.10 and 7.11 show snapshots of the interfaces for creating a new behaviour pattern. The steps to be accomplished are shown as tabs at the top of the interfaces. Figure 7.10 shows the interface to specify the context situation whose fulfilment will trigger the execution of the pattern. Specifically, the figure shows a snapshot of the context situation specification of the *WakingUp* pattern. The working area shows the specification of its two conditions: 1) the *time* property equals to 07:50 and 2) the *workingDay* property equals to true.

Once the user has specified the context situation of the pattern, s/he must navigate to the task specification step, which is shown in Figure

7.11. The working area allows users to specify the tasks that the pattern must execute. To achieve this, a table with the following columns (one for each property of a task) is shown: 1) *Order*, which provides a clearer view of the order in which the tasks will be executed; 2) *Name of the task*, which contains a representative name for the task; 3) *Service*, which contains the name of its related pervasive service (selected on the right frame) and the attributes needed to execute it (which are requested from the user when the service is selected); and 4) *Wait*, which represents the temporal constraint or the context conditions that must be accomplished to execute the next task (its value is 0 by default; if the cell is empty, only one of the tasks will be executed as in the exclusive refinement).

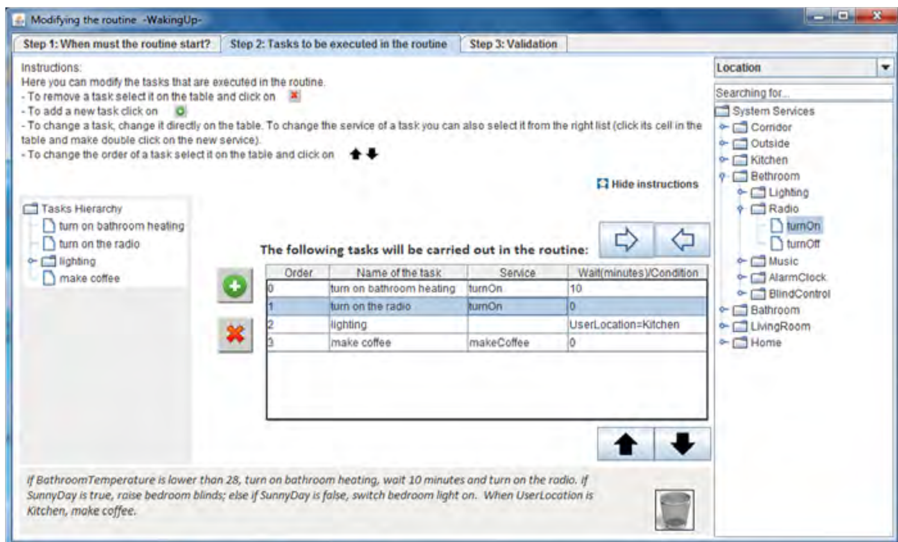

Figure 7.11: Snapshot of the end-user tool for specifying the tasks of a pattern

The operators needed to add a new task or to delete a task selected from the table are shown on the left. When the user adds a new task, a new row in the table is added to allow users to specify the task properties. The *Order* column is auto-completed, although the user

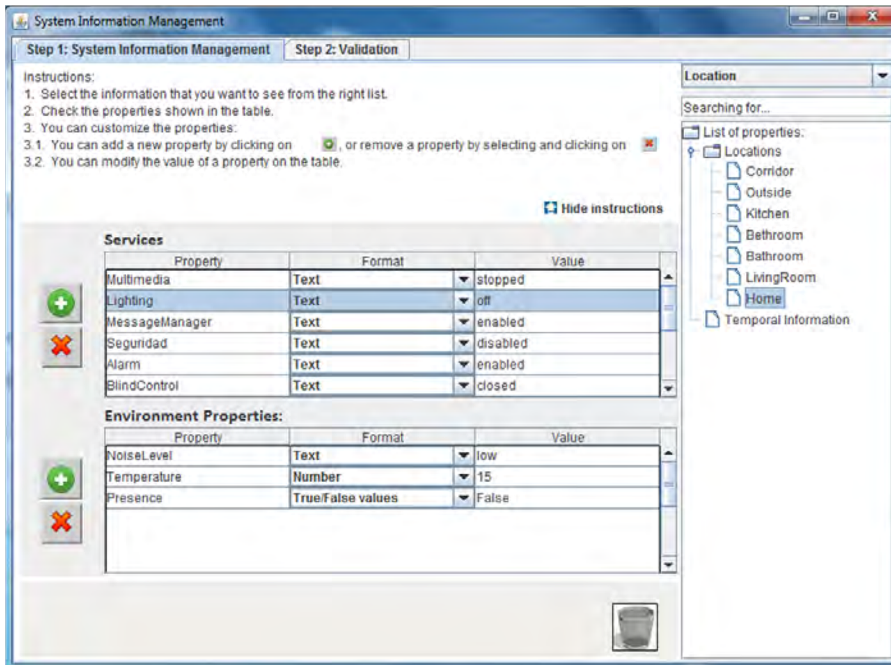

Figure 7.12: Snapshot of the end-user tool for managing context information

can change the order of tasks using the vertical arrows displayed in the working area. The *Service* column only has to be specified if the task is simple (i.e., there is a service that can carry it out); if the task is complex, the user must break it down into simpler subtasks until they can be executed by a provided service. To create a new subtask, the user can use the ⇒ operator, which creates a new row in the table forming a hierarchical form. To return to the parent task level, the user can use the ⇐ operator. As an example, Figure 7.11 shows the tasks that are executed in the *WakingUp* pattern. At the bottom of the interface the explanation of the behaviour pattern is shown as follows: *When time is 7:50 and workingDay is true, if bathroomTemperature is lower than 28, turn on bathroom heating, wait 10 minutes and turn on the radio. If sunnyDay is true, raise bedroom blinds, if sunnyDay is false, switch*

on bedroom lights. When UserLocation is Kitchen, make coffee.

With regard to the context management functionality, as said, it is divided into System Information Access and User Information Management.

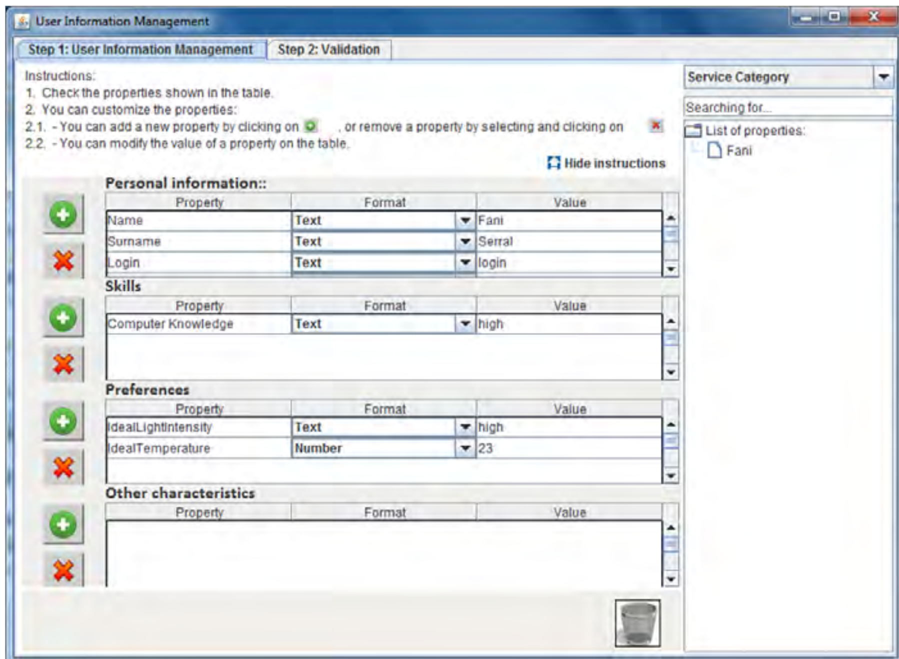

Figure 7.13: Snapshot of the end-user tool for managing user information

Figure 7.12 shows a snapshot of the interface developed for managing the system information. It shows the user the context information related to time, the environment and the system services for which the user has permission. Figure 7.13 shows the interface developed for managing the user information. It allows the user to see his/her information (personal information, preferences, etc.) and to modify it. As shown in the figures, a general view of the information is displayed on the right in a tree representation. In the working area, the properties of the selected context information are detailed. These properties can

be managed using the available operations *add* and *delete*, which are shown with representative images. Using these interfaces, the end-user can, for instance, see the state of the home services or easily change the value of his/her preferences.

In all the interfaces, the last step is always the validation step. The user must navigate to the validation step so that the changes are validated before they are applied to the system. This step is essential to preserve software quality characteristics. Up to date, this validation focus on fulfilling the metamodel constraints (see Section 5.2.3) and checking data type matching in comparisons. If any problem is found, the system notifies the user about the possible mistakes so that they can be corrected. In the future, we plan to extend this tool for also checking that there are no loops in the execution of the patterns and there are no inconsistencies with other patterns. To achieve this, the services provided by the AmI system should provide information about which operations perform contradictory tasks and also about the context properties that each service operation modifies to know if its execution can cause the execution of other patterns. Furthermore, we plan to provide the evolution tool with simulation capacities so that user can simulate the execution of the changed behaviour patterns to check beforehand if these patterns actually do what they want. More detail about this further work will be explained in Chapter 9.

Finally, if the validation is successful, the tool updates the task model and the context model according to the changes made by the users. The tool uses MUTate and OCean to do this at runtime.

7.3.3 Evolving the Behaviour Patterns

Using the same examples described in Section 7.2, we next explain the different ways in which a user behaviour pattern can be modified using the presented tool. Evolutions for creating new behaviour patterns or deleting them are performed in an analogous way.

Evolving the executed services. Figure 7.14 shows how the executed services of the *WakingUp* behaviour pattern are changed to fulfil the new user requirements. As shown, a new task named *inform*

The following tasks will be carried out in the routine:

Order	Name of the task	Service	Wait(minutes)/Condition
0	turn on bathroom heat...	turnOn	10
1	turn on the radio	turnOn	0
2	lighting		BobLocation=Kitchen
3	make coffee	makeExpresso	0

Updating the task

Removing the task

The following tasks will be carried out in the routine:

Order	Name of the task	Service	Wait(minutes)/Condition
0	turn on bathroom heating	turnOn	10
1	turn on relaxing music	turnOn	BobLocation=Kitchen
2	make coffee	makeExpresso	0
3	inform about the current weather	inform	0

Creating a new task

Figure 7.14: Evolving the executed services using the end-user tool

about the current weather has been created in the pattern and has been related to the make coffee task using a temporal relationship of the *enabling* type (wait column with the value 0). The lighting task has been removed. This has automatically removed its subtasks and made that the relationship of the previous task takes the value of the relationship of the removed task (i.e. the make coffee task is still performed when Bob is in the kitchen). The turn on the radio task has been modified to turn on relaxing music by changing the name of the task and the service that executes it, which turns on relaxing music now.

Modifying context conditions. Figure 7.15 shows how the context conditions of the specified behaviour patterns are changed. At the top, the interface for managing the user information is shown. Using this interface, a new user property named *Sick* has been created. This property has been used to change the context situation of the pattern adding a new condition that makes the pattern be enable only when the user is not sick (see in the figure the table where the context conditions are specified). Also, the time of the first context condition has been

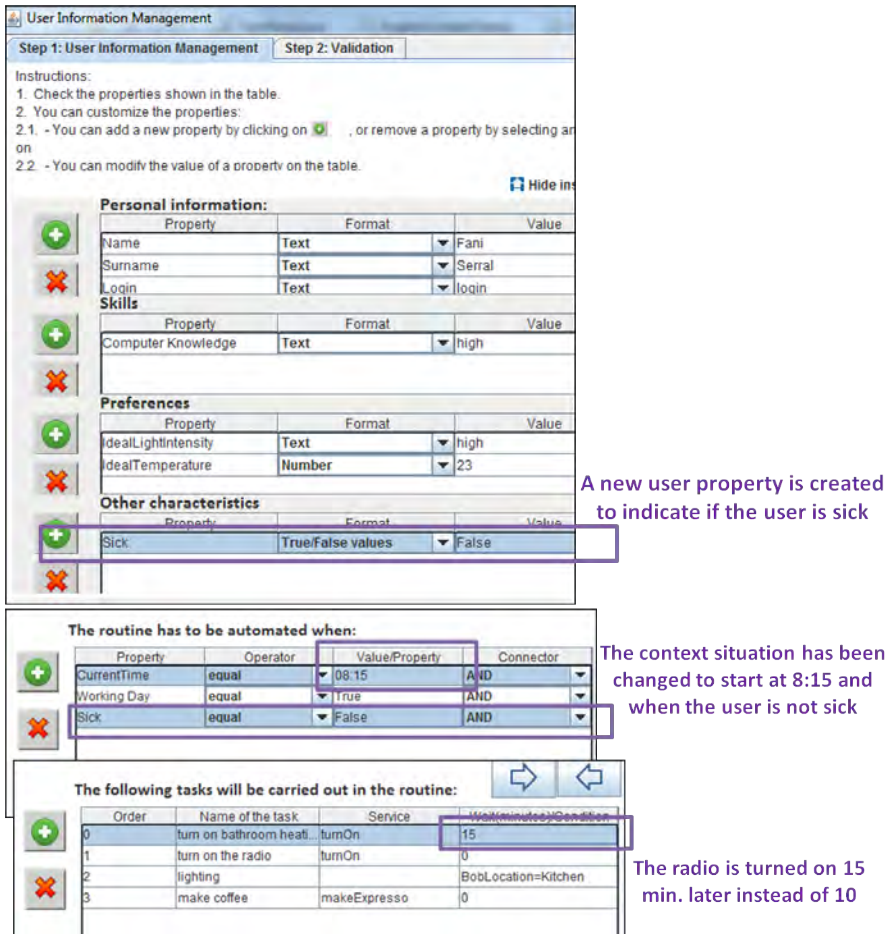

Figure 7.15: Modifying the context conditions using the end-user tool

changed to 8:15. Thus, the pattern starts at 8:15 in working days when the user is not sick. In addition, at the bottom of the figure, it is shown how the temporal restriction of the relationship between the two first tasks of the pattern has been changed to 15 min.

Evolving the execution order of the services. Figure 7.16 shows how the service execution plan of the specified behaviour pattern

is changed. As shown, the last task is now executed the first task, and the first task in now executed the last task. This change has been easily performed using the up and down arrows that allow the execution order of a task to be changed. The relationships of these tasks have been also updated so that each task is executed when the previous one finishes. In addition, the context situation of the pattern has been changed because the pattern must start when the user must be woken up.

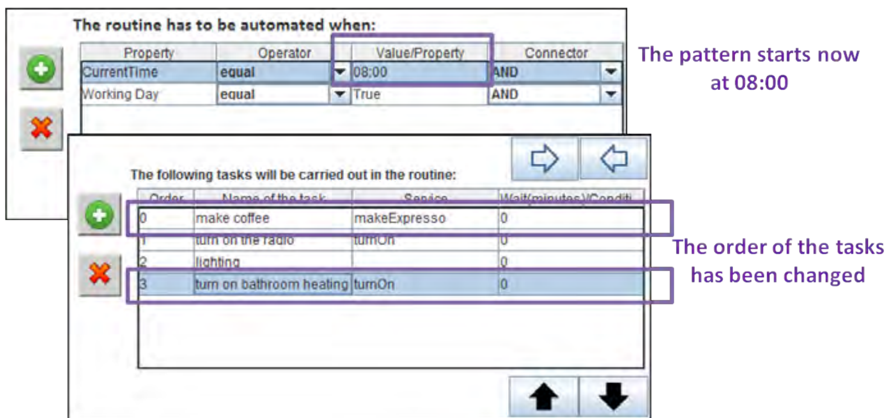

Figure 7.16: Evolving the service execution plan using the end-user tool

7.4 Conclusions

In this chapter, we have explained how the user behaviour patterns specified in the context and task models can be evolved at runtime.

Specifically, we have described the type of evolution that is confronted in this thesis. Next, we have explained how OCean and MUTate, which are the model management mechanisms, can be used to evolve the behaviour patterns. Finally, we have described the designed evolution tool that facilitates the evolution of the specified behaviour patterns by providing intuitive graphical interfaces.

Chapter 8

Evaluation of the Approach

This chapter describes the evaluations performed for validating the approach presented in this thesis. According to the confronted research goals (see Section 1.3), we want to validate the following contributions:

Models. The task model and the context model must provide enough expressivity to specify the behaviour patterns that users want to be automated. Also, this specification must be understandable enough for the end-users so that the models become artefacts for discussion between analysts and users. Thus, the models can properly represent the behaviour patterns to be automated, and do this according to users' demands and desires. Furthermore, the models must allow the behaviour patterns to be specified in a precise way in such a way that they can be automated by directly interpreting the models.

Automation Infrastructure. The provided infrastructure must automate the behaviour patterns in the opportune context as

specified in the models by interpreting them at runtime.

Evolution of the Automated Behaviour Patterns. The mechanisms and tools provided for evolving the automated behaviour patterns must allow them to be easily evolved at runtime according to users needs.

In order to validate these contributions, we put the approach in practice carrying out a case study based evaluation. The philosophy of our approach can be used in any domain where routine tasks are carried out. In this work, we put into practice the approach in several smart home case studies and a nursing home case study.

Since it has been proven that case studies provide deeper understanding of the phenomena under study if proper research methodology practices (Flyvbjerg, 2007; Lee, 1989) are applied, to develop the case studies, we follow the research methodology practices provided in (Runeson & Höst, 2009). These practices describe how to conduct and report case studies and recommend to design and plan the case studies before performing them.

Next two sections describe the pursued evaluation by developing the smart home case studies and the nursing home case study, respectively. Following the recommended reporting structure, each on of these section explains the design and results of the case studies' performance, and provides a summary of the extracted conclusions. After these two sections, we also validate the scalability of our approach, which is a critical problem in pervasive computing (Satyanarayanan, 2001). Finally, we conclude the chapter.

8.1 Smart Home Case Studies

The overall purpose of the development of smart home case studies is to improve users' lives and saving energy resources by automating daily tasks of the users. We applied our proposal to 14 smart home case studies. Following the reporting structure, we first describe the design

of these case studies. Then, we describe the results obtained from them. Finally, we explain the conclusions extracted from the evaluation.

8.1.1 Design of the Smart Home Case Studies

In order to design the smart home case studies, we follow the guidelines provided by Runeson and Höst in (Runeson & Höst, 2009). According to the reporting guidelines, we first determine the research questions that we want to validate. Second, we describe the case and subject selection. Third, we determine the selected procedures to collect the data. Fourth, we explain the procedures and techniques used to analyse the collected data. Finally, we describe the validity procedures.

Research Questions

By developing the smart home case studies we want to evaluate the following research questions:

1. Do the models provide enough expressivity to describe the behaviour patterns that the users of the case studies want to be automated?
2. Are the models understandable for the users involved in the case studies and useful for discussing the specified behaviour patterns?
3. Does the infrastructure correctly automate the behaviour patterns as specified in the models by directly interpreting them?
4. Are the provided mechanisms and the graphical evolution tool useful for evolving the specified behaviour patterns?

Case and Subjects Selection

In a case study based evaluation it is important to use several data sources in order to limit the effects of one interpretation of one single data source. It is recommended to take into account viewpoints of

different roles or personalities, or develop several case studies of the same characteristics. If the same conclusion can be drawn from several sources of information, this conclusion is stronger than a conclusion based on a single source.

Thus, to validate these research questions, we selected 14 smart home systems covering different number of inhabitants (families, couples and single people). We selected smart homes because their development is an issue in which industry is very interested. This is because home is a fertile ground for offering products and services to improve the lives of people.

The subjects of the evaluation were the clients of the case studies. We selected them taking into account that they covered a wide variety of professions, ages, intellectual capacities, studies, and computers' knowledge.

Data Collection Procedures

Before starting the process of collecting data, the participants completed a questionnaire with a few questions to find which vocabulary they use to describe a behaviour pattern. The questions asked for the definition of the task concept first. Then, we asked them to determine how they refer to a set of tasks habitually performed. Finally, questions were asked for the definition of other important concepts such as condition, service or context.

We then prepare and set up the data collection within the following four steps:

Step 1: Identify the behaviour patterns to be automated. To perform this step, we designed a semi-structured interview to know the behaviour patterns that the subjects wanted to be automated by the system. A semi-structured interview is composed of planned questions, but they have not to be necessarily asked in the same order as they are listed. Thus, we could decide in which order the different questions should be handled according to the development of the conversation in the

interview. Also, we could use the list of questions to be certain that all questions were handled. Additionally, semi-structured interviews allow for improvisation and exploration of the studied objects; therefore, we could improvise more questions if needed.

The process for identifying the behaviour patterns is not a contribution of this work. Thus, in this section we only summarize this process; more detail information about its design and results can be found in Appendix B.

Step 2: Specify the models and check their comprehensibility by users. After identifying the behaviour patterns that had to be automated, we specified them using the task and context models. After teaching the subjects about the main components of the task model notation, we refined the models until the subjects agreed with the behaviour patterns specified. In this process, we checked the comprehension of the models by the subjects. To do this, we used a short semi-structured interview. According to (Gemino & Wand, October 2003), in order to evaluate the understanding of a modelling technique, tasks that require reasoning about the models are needed. Thus, we asked questions to the subjects that make them reason about the task model. For instance, some of these questions were: how many tasks will be executed in this routine?; when will this routine be activated?; when will this task be executed?.

Step 3: Testing the provided software infrastructure. Once the models were validated, we developed the services that support the functionality needed to execute the system tasks of the patterns.

Next, we put the system into operation. To do this, we used a scale environment with real devices[1] to represent the Smart Home. This execution environment is made up of a PC and a network of KNX devices[2] connected to the PC by a USB port. In the PC, an Equinox distribution (which is the OSGi implementation

[1] http://oomethod.dsic.upv.es/labs/projects/pervml
[2] http://www.knx.org

of Eclipse) was run. The pervasive services, MAtE, MUTate, OCean and the graphical evolution tool were installed and started in Equinox. Since the device network did not provide us with all the needed devices for the case studies, we also used a simulator for simulating the behaviour of the rest of needed devices. This simulator was presented in (Muñoz *et al.*, 2005) and allows us to define virtual devices and control them using an intuitive user interface. In addition, the specified models were copied in the folder were Equinox was installed.

Once the system was running, we used the device simulator to simulate the fulfilment of the context situations specified in the task model. Then, we checked using JUnit tests, whether the behaviour patterns were automated as they were specified in the models. We repeated this simulation after evolving the automated behaviour patterns using OCean and MUTate and the graphical tool. Some of the evolutions that were performed have been explained in sections 7.2 and 7.3.

Step 4: Usability evaluation of the graphical tool. We made the subjects of the case studies perform a series of scenarios using the graphical evolution tool. To do this, we first designed the scenarios that should be performed. We then arranged several sessions in which the users carried out these scenarios under our supervision. In these sessions, we grouped the users according to their level of computer knowledge. Thus, before performing the scenarios, we could explained them how the tool works according to their computer knowledge (using a few minutes for explaining the tool to those users with high computer knowledge and a detail explanation for those users with very basic computer knowledge). In any case, we did not expend more than half an hour to explain them the tool.

Some examples of the evolutions that they performed were described in sections 7.2 and 7.3. In total, the scenarios that they performed included:

- delete, modify, and add a context property;

- enable/disable and delete a behaviour pattern
- modify a behaviour pattern by changing its context situation and its tasks (service in charge of executing them, temporal relationships, etc.);
- add a new simple behaviour pattern (without composite tasks): create the context situation and specify the tasks to be executed;
- add a new complex behaviour pattern (with composite tasks): create the context situation and specify the tasks from more complex to simpler.

Once the users completed the scenarios, they sent us the result task and context models so that we can check whether the users had correctly performed the scenarios. Then, they filled out the Post-Study System Usability Questionnaire (PSSUQ) published by IBM in (Lewis, 1995). This questionnaire is a 19-item instrument for assessing user satisfaction with system usability. Specifically, it allows us to measure the overall satisfaction (OVERALL) of the system, its usefulness (SYSUSE), its information quality (INFOQUAL), and its interface quality (INTERQUAL). In addition, we extended this questionnaire to know: general information about the users (such as the age of the subjects, their computer knowledge, etc.); how long users take to perform the scenarios; whether participating in the model specification helps them to use the tool or not; and their general opinion about the interfaces (what they like about the interfaces, what they would change, what is the most difficult step to perform, etc.).

Analysis Procedures

The analysis of the obtained data was conducted as follows in order to answer the established research questions:

- The first research question was whether models provide us with enough expressivity to describe the behaviour patterns that the

users of the case studies want to be automated. To answer this question, we analysed the behaviour patterns identified to be automated, the obtained models for specifying them and our experience using the models. Using these artefacts, we analysed whether the models could properly represent all the needed behaviour patterns.

- The second research question was whether the models were understandable by the users involved in the case studies and useful for discussing the specified behaviour patterns. To answer this question, we use our experience in the model refinement with the end-users and analysed the results of the questionnaires in Step 2 of the data collection, obtaining statistical data from them.

- The third research question was whether the infrastructure correctly automates the behaviour patterns as specified in the models by directly interpreting them. To answer this question, we analysed the results obtained from Step 3 of the data collection.

- The fourth research question was whether the provided evolution mechanisms and the graphical tool were useful for evolving the specified behaviour patterns. To answer this question, we analysed the results obtained in Step 3 of the data collection. We also analysed the results of the questionnaires performed in Step 4, obtaining statistical data from them.

Validity Procedures

The selection of case studies for different number of inhabitants and the selection of participants with a wide variety of personalities, capacities, etc., maximize the external validity of the results.

In addition, we presented and discussed the results of each interview and questionnaire with the corresponding subjects of the case studies and also discussed the results of the overall analysis with other researchers.

8.1.2 Results of the smart home case studies

To develop the smart home case studies, we follow the process proposed in this thesis, which is described in detail in Chapter 4: first, we identified the behaviour patterns to be automated; we then specified the identified behaviour patterns using the proposed context and task models; using these models and the provided software infrastructure, we ran the system; finally we evolved some of the specified behaviour patterns at runtime by using the provided mechanisms and tools.

To explain the results of the development of the smart home case studies, we first describe the involved participants, and then organize the section according to the steps designed in the data collection procedures. In each one of these steps we analyse the data following the designed analysis procedures, which allow us to answer the proposed research questions.

Subjects' Description

The involved participants were: 3 families (one family with two little kids and two families with 1 kid), 1 couple and 10 single people. They had a wide variety of professions including 10 engineers, an administrative, a teacher, a housewife, two nurses, a farmer, a carpenter and a student. Since the kids were too little to be able to participate in the development of the case studies, they were represented by their parents. Finally, the total of participants was 18 which ranged from 26 to 57 years of age. From them, 10 were men and 8 women. Also, those that were engineers have a medium-high level of computers' knowledge, while the rest only have basic computer knowledge.

Identify the Behaviour Patterns to be Automated

Before identifying the behaviour patterns to be automated, the subjects of the case study completed the prepared questionnaire to find which vocabulary they used to describe a behaviour pattern. The questions asked for the definition of the task concept first. As stated in the

literature (Johnson, 1999; Paternò, 2002), this term was perfectly understood by them. They defined a task as work to be done or work that has to be performed habitually, like it is described in the common definition of task. Then, we asked them to determine how they refer to a set of tasks habitually performed. Most of them, over a 83% use the term *routine*, and the rest use the term habit or daily tasks.

Then we followed the designed semi-structured interview to know which behaviour patterns users want to be automated. We analysed the provided information and identified the behaviour patterns that could be useful for the users. We identified from 6 to 12 behaviour patterns to be automated in each case study, with a total of 97 behaviour patterns. In essence, from these behaviour patterns we detected 15 that were different (i.e., those that had different goals). The rest of behaviour patterns were variations of them. For instance, most users wanted automatically room lighting taking into account outside light intensity and user presence. If the user slept alone, he or she wanted the room was always illuminated, however, if users were a couple or have babies, they usually wanted the light in the bedroom was not switched on when there was someone sleeping. In contrast, another couple wanted the lights were not switched off while her daughter was playing in the house.

More detail information about the results of these interviews can be found in Appendix B.

Specify the Models and Check their Comprehensibility by Users.

We specified the identified behaviour patterns using the context and task models. The context needed for automating the identified behaviour patterns was specified in the context model. Figure 8.1 shows part of a context model specified for one of the case studies. The behaviour patterns were specified in the task model using the context of the context model. We next describe some representative examples of these patterns. Their specification using the task model is shown in Figure 8.2. In the behaviour pattern examples, we use false names to

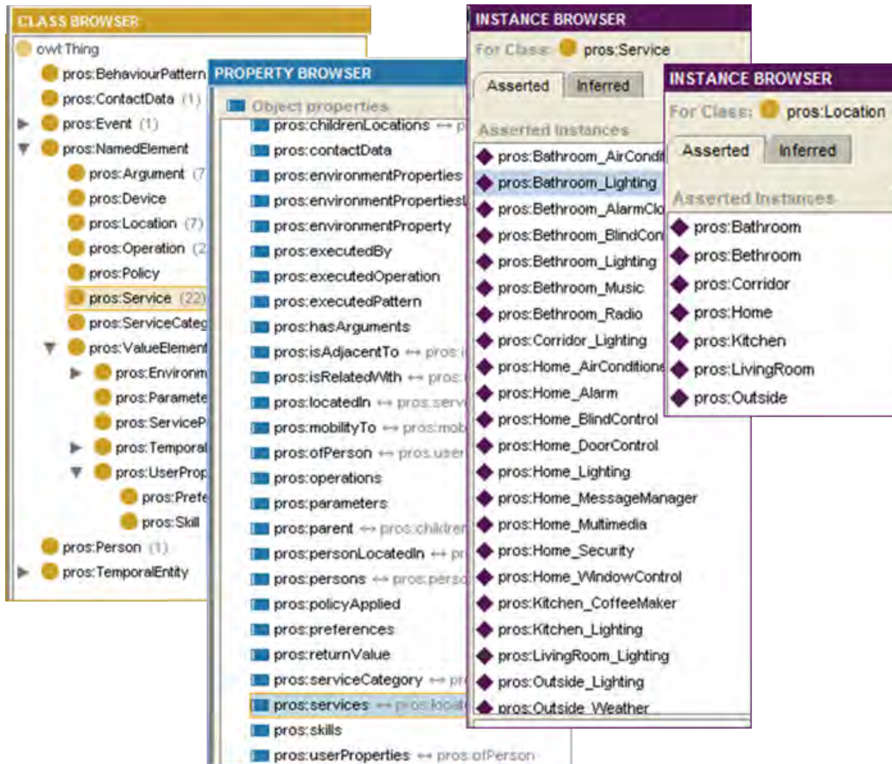

Figure 8.1: A context model examples of the smart home case studies

keep user privacy:

1. **Waking up**: At 6:50 a.m. on working days, the system switches on the bathroom heating. 10 minutes later, the system puts the radio quietly on in the bedroom to wake up Bob. At 07:30 a.m., the system puts the radio again to wake up Sarah. Afterwards, when the users are in the kitchen to have breakfast, the system recommends them the best transport to go to work.

2. **Airing home**: The first time users leave home, the system airs the house. To do this, the system raises blinds and opens windows.

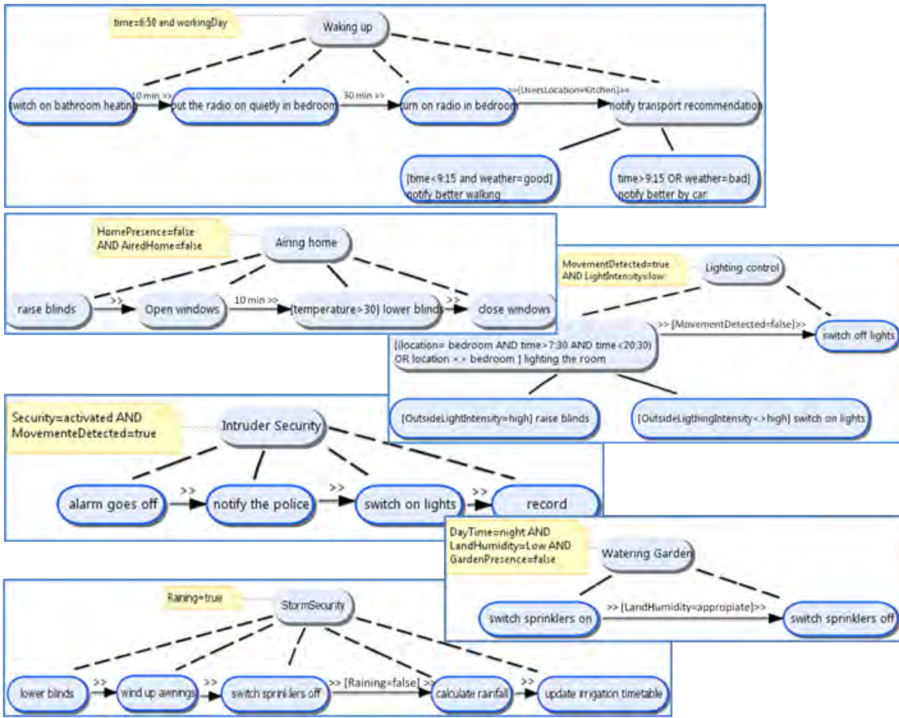

Figure 8.2: Examples of the models specified in the smart home case studies

After 10 minutes, the system lowers blinds if the temperature is greater than 30°C, and closes windows.

3. **Lighting control**: When movement is detected in a room and its light intensity is low, the room is illuminated provided that it is not the bedroom. If it is the bedroom, it is only illuminated if it is between 7:30 and 20:30. For illuminating a room, if it is a sunny day, the system raises the blinds; otherwise, the system switches on the lights. Afterwards, when movement is not detected any more, the lights are switched off.

4. **Intruder security**: If the security system is activated and an intruder is detected, the system makes the house alarm go off,

notify the police, switches on the lights to simulate presence, and starts to record by using the installed cameras.

5. **Watering garden**: At night, when the land humidity is low and nobody is in the garden, the sprinklers are switched on. When the proper land humidity is achieved, the sprinklers are switched off.

6. **Storm security**: If it starts to rain, the system lowers all the blinds, winds up all the awnings and switches off the garden sprinklers. When it stops raining, the system calculates the cubic meters of rainfall and updates the irrigation timetable according to it.

All the identified behaviour patterns to be automated could be specified using the proposed context and task models; however, although the following aspects could be represented, the way in which them had to be specified was not very direct and intuitive:

- Perform some task only a certain number of times at day, a certain number of times at week, etc. This aspect can be modelled by adding a context property in the context model that counts the times that has been executed the task. This property is used in context conditions of the task model to control the times that the task must be carried out. An example of the specification of this aspect is shown in the behaviour pattern *Airing home* in which the AiredHome context property is used so that the pattern is executed only once (see Figure 8.2).

- Perform a task during a period of time. In our approach, system tasks are atomic tasks and we consider that they do not have a duration (i.e., we consider them as stateless tasks). For this reason, to specify that a task has to be executed for a period of time, we specify the start of the task and its end, relating both events using a temporal relationship with a temporal restriction equivalent to the duration that the task must have. For instance,

in the behaviour pattern *Airing home*, windows are opened during 10 minutes (see Figure 8.2).

- Loops. Tasks such as switching lights or raising blinds in a gradual manner can be specified by several tasks in which the argument was higher and higher (e.g., with light intensity 100, 200, 300, etc.). These tasks would be related by temporal relationships with the needed wait time (e.g., 1 min, 2 min, etc.).

Once specified the identified behaviour patterns, we validated them with the end-users. To do this, we first taught the users about the main concepts of the task model, and then checked the model comprehension using the prepared short questionnaire. This questionnaire made users reason about the model. We found that 14 from the 18 users perfectly understood the behaviour specified in the task model. The other 4 users, those with a lower level of studies, understood very well the structure of the model (task hierarchy and task relationships); however, they had difficulties to know what meant the used context conditions. To solve this problem, we describe them in natural language. We eliminated the task preconditions adding this information to the name of the task (e.g., instead of specifying *[LandHumidity=low] switch sprinklers on*, we specified: *if the humidity of the land is low, switch sprinklers on*) and also replaced the $>> [c] >>$ temporal relationship with text (e.g., instead of specifying $>> [UsersLocation = Kitchen] >>$, we specified *when users arrive to the kitchen*).

Afterwards, we explained to the users the specified behaviour using the model, which results very useful to discuss and validate the behaviour patterns to be automated. If something was not specified as users wanted to be automated, we refined the model to fulfil their requirements. We repeated this process until the users agreed with the patterns specified in the task model.

Testing the Provided Software Infrastructure

To support the functionality needed to execute the system tasks of the patterns, we obtained the code of the required services by using the

MDD strategy presented in (Muñoz *et al.*, April 2006; Serral *et al.*, 2010). We had already many of these services developed from previous case studies (Muñoz *et al.*, April 2006; Serral *et al.*, 2010), therefore, we could reuse them. At the end, we had a total of 26 different services. More detail about them can be found in Appendix B.

We then evaluated the feasibility of our approach. Using the running system, we validated that the behaviour patterns were automated as they were specified in the models. Specifically, the following aspects were validated before and after evolving the behaviour patterns:

- All the behaviour patterns are triggered only when its context situation is fulfilled.
- When a behaviour pattern is executed, all the required services are executed in the correct order and in the correct conditions.

To perform this, we based on the fact that the Context Monitor registers in the ontology each execution of a service (see Section 5.1.3 and 6.3.2). Thus, the proposed validation consisted in: (1) simulating the fulfilment of specific context conditions in order to trigger the execution of several behaviour patterns, and (2) checking that all the services that must be executed were registered by the context monitor in the correct order, respecting the corresponding temporal relationships between the tasks. Before performing this validation, we validated: that MUTate and OCean properly retrieved and saved data (see Section A.1 for more detail), and that the Context Monitor registered service execution in a proper way.

In order to perform all these validations, we used simulations and JUnit tests[3]. We developed a set of JUnit tests that allow us to evaluate the behaviour of OCean and MUTate, and also MAtE and the Context Monitor. For instance, as a representative example, Figure 8.3 shows the JUnit method that evaluates the behaviour pattern execution. This method compares the tasks executed when a behaviour pattern is triggered (i.e., those registered by the context monitor) with the real

[3]http://www.junit.org/

```
public void TestExecuteBehaviourPattern(BehaviourPattern bp){

  List<Task> executionPlan=bp.getExecutionPlan();

  executeBehaviourPattern(bp);

  List<AutomaticOperation> automatedOperations= contextModel.getLastAutomaticOperation
                                                (executionPlan.size());

  ArrayList<String> plannedTasks, executedTasks;
  for(Task t: executionPlan) planedTasks.add(t.getID());
  for(Operation o: automatedOperations) executedTasks.add(o.executedOperation.getID());

  assertEquals(plannedTasks, executedTasks);
}
```

Figure 8.3: JUnit test for evaluating that all the pattern tasks are executed

execution plan of that behaviour pattern (i.e., the tasks that should be executed).

To perform this evaluation, we implemented the getExecutionPlan(id) method. It returns the execution plan of the behaviour pattern whose ID is the received id. This execution plan is a list of names of the system tasks that will be executed according to context. We manually initialized the executed plan of 15 different behaviour patterns (i.e., one for each different goal), which were a representative set to test that the behaviour patterns were properly executed.

After executing a pattern, we retrieved the last registered automated operations from the context model (i.e., we retrieved the individuals of the AutomaticOperation class) by using OCean. We retrieved as many automated operations as tasks the obtained execution plan has. Finally, we created an equal assertion to check if the automated operations retrieved from the context model were the same as the tasks that contained the execution plan.

This JUnit test was executed for the 15 selected behaviour patterns after simulating the context conditions in which each pattern should be executed in the same manner than its manual introduced execution plan.

After evaluating that the behaviour patterns were correctly automated, we evolved them according to the planned evolutions. For

each evolution, we applied again the JUnit tests and checked that the behaviour patterns were correctly executed according to the performed evolution. We performed these evaluations in an iterative way, which allowed us to detect and resolve some mistakes. For instance, we realized that the behaviour patterns dependent on time, made the system enter in a loop. This was because the system updates time every second and the smallest time unit considered in the behaviour patterns was minutes. Thus, the context situation of these patterns was continuously fulfilled until a minute went off. To solve this problem, we needed to use the same time unit in both cases. Considering that updating each second the context model could overload the system, we updated the context monitor so that the time was updated every minute.

Usability Evaluation of the Graphical Tool

The complete PSSUQ questionnaire results are shown in Figure 8.4. According to the results of the questionnaire, the tool received the following ratings on a scale of 1 (the highest score) to 7 (the lowest score): overall satisfaction was 2.158, usefulness was 2.041, information quality was 2.056, and interface quality was 2.092. These results revealed that although some aspects of the tool have to be improved, it was clear enough and simple to use for most of the subjects of the case studies, allowing them to evolve the behaviour patterns.

In the arranged sessions, all the subjects could correctly perform all the evolution scenarios without much difficulty. This perception was also observed by the users as the answers to the question 3 of PSSUQ shows. Also, the users noticed that they did not need too much time to perform the scenarios (see the answers to the questions 4 and 5 of PSSUQ). For instance, the first scenario was to create a simple routine with two conditions and two tasks, which took the users from 4 to 10 minutes. The last scenario was to modify all the aspects of a created behaviour pattern. Although the latter was a longer scenario, it took the users from 30 seconds to 4 minutes. We observed and validated with the users that the first scenario took more time than the others because the users were also understanding how the tool worked; however, the results

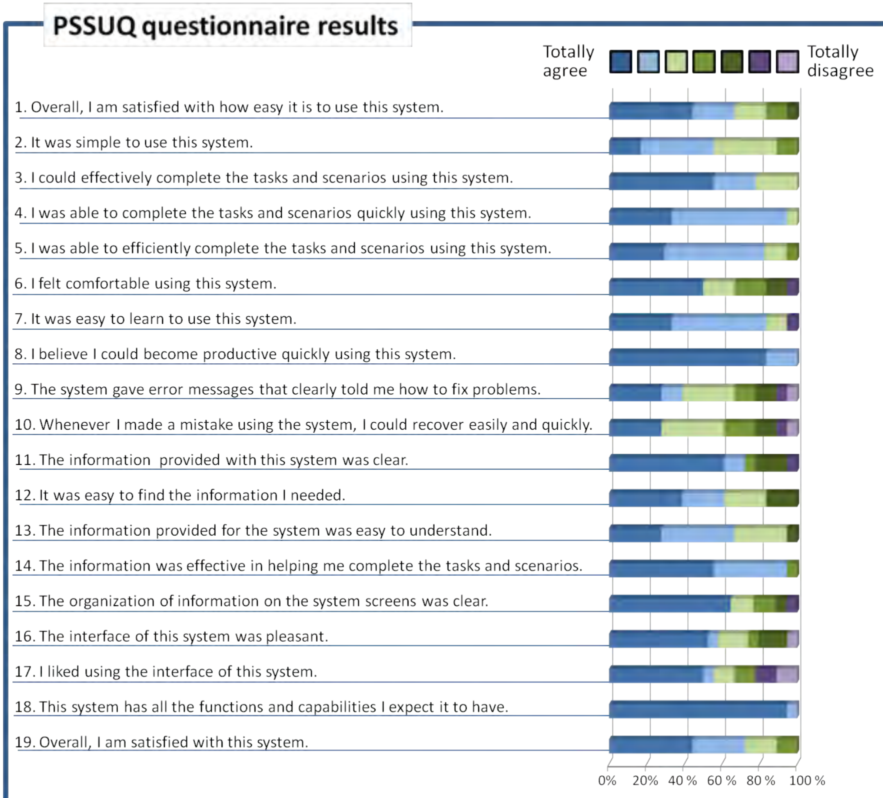

Figure 8.4: Results of the PSSUQ Questionnaire

revealed a tendency to need less and less time to perform the scenarios. This is because all the interfaces of the tool follow a similar structure and provide users with all the needed functionalities and information (see the answers to the questions 14, 15 and 18 of PSSUQ), which helped the users to quickly be more efficient. In fact, all the users believed that they could become productive quickly using the tool (question 8 of PSSUQ) and most of them answered that it was easy to learn to use the tool (question 7 of PSSUQ).

The worst score obtained from the questionnaire was for the

questions 9 and 10, which determine the information quality of the message errors. Users commented that the help messages that were shown in the interfaces helped them to complete the tasks; however, when they committed an error, the error messages were not clear enough to correct them. To improve this aspect, we are currently working to make these messages clearer for users by showing examples of solutions to correct the possible errors.

Regarding the participation in the task model specification, the 72% of the users commented that this process helped to familiarize them with the tool, and above all, facilitate them to modify the automated patterns and create new ones because they already knew how they were structured.

Regarding what users did not like or they would change, some users, essentially those with a low level of studies and people older than 50 years old, commented that designing the context conditions was still difficult. They commented that they would like to have a list of predefined conditions in which they could change some parameters (i.e., when it is -day of week-, at -time-, in a working day, when nobody is at home, etc.). After explaining them how a condition is formed in depth, we observed that the interface design helped them to correctly specify the conditions since it provided all the context properties and the operators that they could use and helped the users to fill the values.

In addition, some users also commented that they preferred forms instead of filling out tables, because they said that forms would facilitate to change the automated behaviour patterns. We have extended the tool to support this aspect. We have created forms for adding and modifying a condition, a task and a context property. As examples, Figures 8.5 and 8.6 show the forms created for modifying a context condition and a task, respectively. These forms offer users a better guide for entering the required information. This has allowed us to simplify the interface instructions. Also, the created forms make much clearer the description of context condition unions and task relationships for users that it was in the tables. Furthermore, in the new interfaces, we have substitute the information tree shown on the right for combo trees in the forms. In this way, the place where the tree information must be used is completely

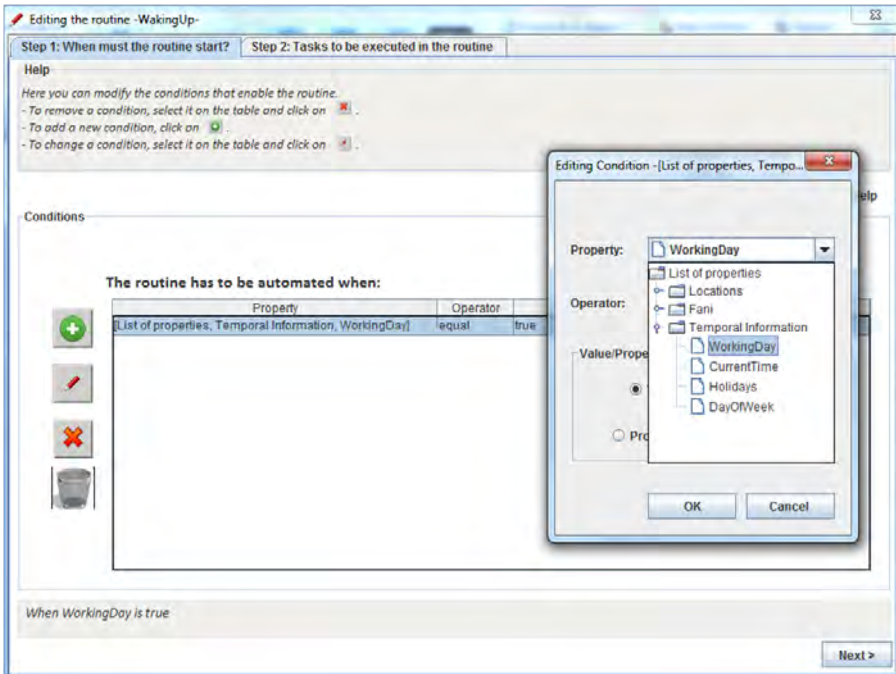

Figure 8.5: Interface extended with forms for modifying a context condition

delimited, considerably facilitating to fill the required information.

Regarding the validation step, many users mentioned that they liked it because it guaranteed them that the changes were correct. However, some users said that they usually forgot to navigate to this step and tried to finish before. Although the tool did not allow users to exit without validating or reject the changes, it needed the user navigated to the validation step. To avoid this, we have improved this aspect by eliminating the validation step placed at the top and moving this funcionality to the save and close button (see Figure 8.7). When users try to exit, a message is shown to ask them whether they want to save the changes or not. If they decide to save them, change validation is automatically done, just like when users save the changes by using the save button. In the example shown, the user is trying to save a routine

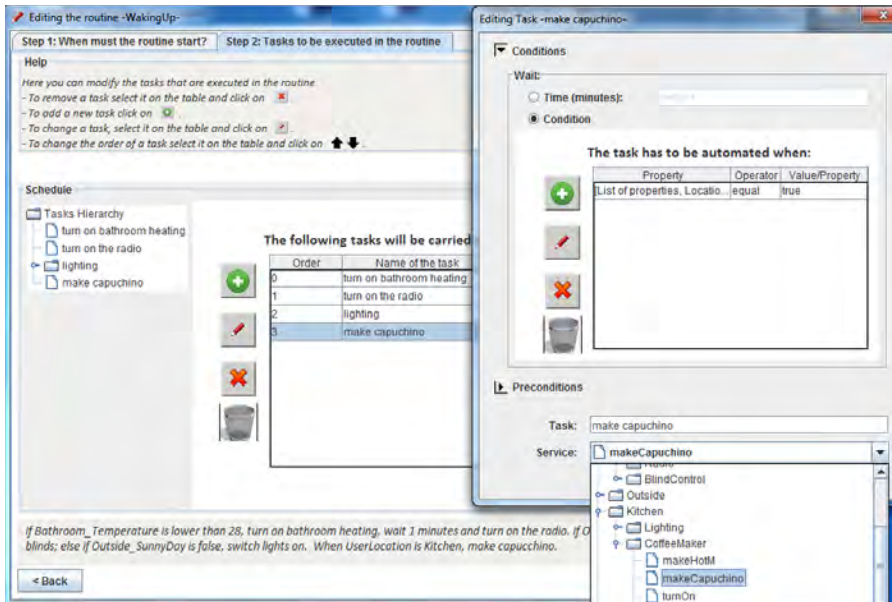

Figure 8.6: Interface extended with forms for modifying a behaviour pattern task

without a context situation; therefore, the system reports her the error and does not let her to save the changes (until the error is solved). Thus, the validation is performed unconsciously by users.

Regarding what users like the most, many users indicated that they liked the functionality that the tool provided them with because it easily allowed them to change the automated tasks. Some users also commented that they liked the structure of the interface because it was very similar over all the interfaces, it was clear and its information was well organized. Also, some users said that they liked the instructions provided in the interfaces because these instructions guided them in the tasks to be performed.

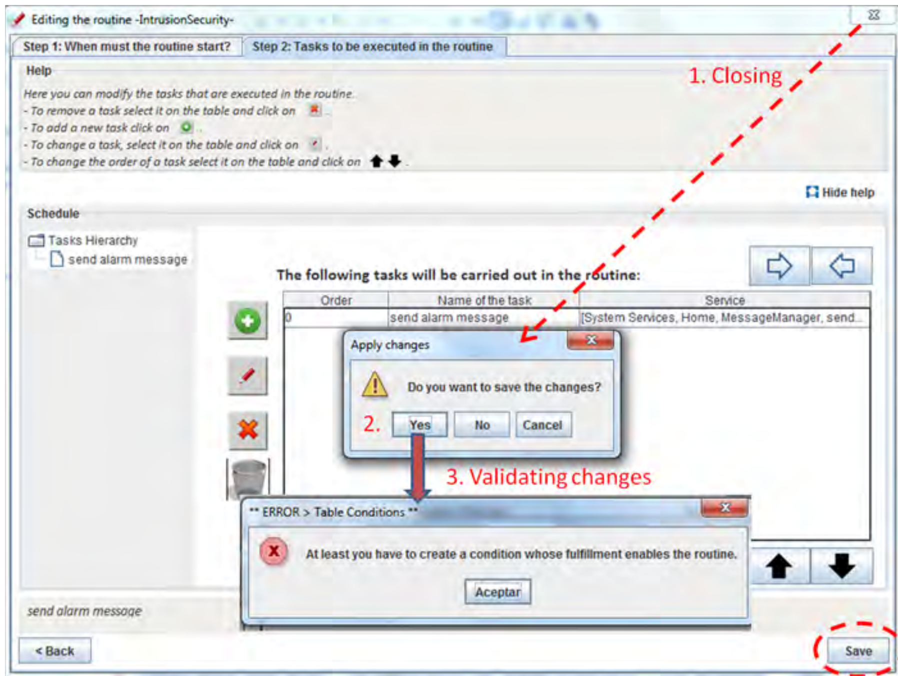

Figure 8.7: Interface that shows the change validation

8.1.3 Conclusions of the Smart Home Case Studies' Validation

The development of the smart home case studies has allowed us to validate the four research questions presented at the beginning of the section.

First of all, we specified the behaviour patterns that the users wanted to be automated. This has allowed us to validate that, although some aspects could be improved, the models proposed in this thesis provide enough expressivity to describe the behaviour patterns needed for all the smart home case studies (*Research Question 1*). In addition, we refined the models with the user participation, checking that these models are mainly understandable by the users and can be very useful for discussing

and validating the specified behaviour patterns with the users (*Research Question 2*).

After validating the models with the end-users, we made some simulations and passed a set of JUnit tests checking that the models are correctly executed by the the provided software infrastructure. We have also validated that after evolving the specified behaviour patterns, the software infrastructure correctly automates all the behaviour patterns applying the corresponding performed evolutions (*Research Question 3*).

Finally, we have validated that the provided tool is useful for the users in order to evolve the specified behaviour patterns according to their needs. However, it needs to be improved for facilitating the evolution by end-users with a low level of computer knowledge or skills (*Research Question 4*).

8.2 Nursing Home Case Study

To validate our approach, we also developed the automations needed for the ACube[4] research project. This project aims at designing an automated user intensive system to be deployed in nursing homes as a support to medical and assistance staff. The ACube consortium has a multidisciplinary nature, involving software engineers, sociologists and analysts, and it is characterized by the presence of professionals representing end-users directly engaged in design activities.

Following the reporting structure, we first describe the design of the case study. Then, we describe the results obtained from it. Finally, we explain the conclusions extracted from this evaluation.

8.2.1 Design of the Nursing Home Case Study

In order to design the nursing home case study, we follow the guidelines provided by Runeson and Höst in (Runeson & Höst, 2009). According to

[4]The ACube project was founded by the local government of the Autonomous Province of Trento in Italy; see http://acube.fbk.eu/en/node/57

the reporting guidelines, we first determine the research questions that we want to validate. Second, we describe the case selection. Third, we determine the selected procedures and techniques to collect the data, and to analyse it. Finally, we describe the validity procedures.

Research Questions

The requirements specifications of the ACube case study were already available (see a detailed explanation of them in Appendix B). This means that the quality of the captured data of this case study is not under our control. However, the purpose of capturing this data was to design an automated system to support and automate the medical and assistance staff tasks. Thus, although we could not actually interact with the subjects during the development of the case study, we consider it a valuable case study for validating the following research questions:

1. Do the models provide enough expressivity to describe the behaviour patterns that automate medical and assistance staff tasks?

2. Does the software infrastructure correctly automate and evolve the specified behaviour patterns?

Case Selection

The main goal of automating the behaviour patterns performed by medical and assistance staff is to help them to make their work more efficiently in order to enhance the quality of life of their patients.

By automating behaviour patterns, the tasks of medical and assistance staff can be greatly reduced freeing them so that they can spend more time with their patients. In addition, the routine tasks that medical and assistance staff perform can be improved to be more efficiently because they can be previously analysed and also can be automated even when none of the staff is present.

Data Collection Procedures

As said, in this case study we cannot directly interact with the users. For this reason, the data collection was performed only by using archival data. Specifically, we used: the requirement specification of the ACube project, the context and task models in which the behaviour patterns that must be automated were specified, and the results obtained by performing different simulations of the system in execution. Thus, we prepared and set up the data collection within the following three steps:

Step 1: Identify the behaviour patterns to be automated. From the requirements of the ACube case study, we identified several behaviour patterns that could be automated to support the automation of medical and assistant staff.

The process for identifying the behaviour patterns is not a contribution of this work. Thus, in this section we only summarize this process; more detail information about the requirement artefacts and how the behaviour patterns were identified can be found in Appendix B.

Step 2: Specify the behaviour patterns to be automated. After identifying the behaviour patterns that had to be automated, we specified them using the task and context models.

Step 3: Testing the provided software infrastructure. Once the models were validated, we developed the services needed to execute the system tasks of the patterns.

We then put the system into operation. We used a PC with an installed Equinox. To run the system, the pervasive services, MAtE, MUTate, OCean and the device simulator (presented in (Muñoz *et al.*, 2005)) were installed and started in Equinox. The models were copied in the folder were Equinox was installed.

Once the system was running, we simulated the fulfilment of the context situations specified in the task model and checked using JUnit tests, if the behaviour patterns were automated as they were specified in the models.

We repeated this simulation after evolving the automated behaviour patterns using OCean and MUTate and the graphical tool. Some of the type of evolutions that were performed have been explained in sections 7.2 and 7.3.

Analysis Procedures

The analysis of the obtained data was conducted in the following steps organized in order to answer the established research questions:

- The first research question was whether the models provide us with enough expressivity to describe the behaviour patterns that have to be automated for the case study. To answer this question, we analysed the identified behaviour patterns (Step 1 of the data collection) and the models specified for automating them (Step 2 of the data collection). Using these artefacts, we analysed if the models could represent the behaviour patterns that were identified in the requirements of the case study.
- The second research question was whether the infrastructure correctly automates and evolves the behaviour patterns specified in the models. To answer this question, we analysed the results obtained from the Step 3 of the data collection.

Validity Procedures

We presented, discussed and validated the specified models with the researchers in charge of performing the requirement elicitation of the ACube project. In addition, we discussed the results of the overall analysis with them and other researchers.

8.2.2 Results of the the Nursing Home Case Study

As we did for the smart home case studies, to develop the nursing home case study, we followed the process proposed in this thesis (see Chapter 4): first, we identified the behaviour patterns to be

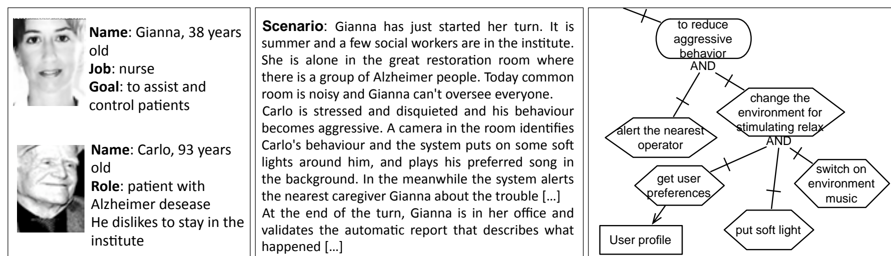

Figure 8.8: Three produced artefacts: a couple of relevant *Personas*, the scenario of *aggressive behaviour* in which they are involved and the slice of correspondent goal model.

automated; we then specified the identified behaviour patterns using the proposed context and task models; using these models and the provided software infrastructure, we ran the system; finally we evolved some of the specified behaviour patterns at runtime by using the provided mechanisms and tools.

To explained the results of the development of this case study, we organize the section according to the steps designed in the data collection procedures. In each one of these steps we analyse the data following the designed analysis procedures, which allow us to answer the proposed research questions.

Identify the Behaviour Patterns to Be Automated

The capture of requirements of the expected system was already done by the Fondazione Bruno Kessler IRST of Trento in Italy (see Appendix B for a detail description of these requirements). We had four scenarios, a set of *Personas* (Cooper *et al.*, 2007) and a Tropos model (Bresciani *et al.*, 2004) at our disposal to obtain the behaviour patterns that must be automated. Figure 8.8 shows examples of these artefacts. Particularly, this figure shows the following:

- A couple of *Personas*. *Personas* are powerful instruments for creating descriptive models of system-to-be users based on

behavioural data. *Personas* are derived from patterns observed during interviews, with the aim of representing the diversity of observed motivations, behaviours, and mental models. Two examples are given in the figure: Carlo, who is a patient with Alzheimer disease, and Gianna, who is a nurse.

- A technological scenario. Technological scenarios are short narrative stories that represent people (Personas) acting in a specific context and supported by the envisage technology. They increase the insight on requirements of the system but also are useful for communicating and validating requirements with non-technical people. The figure shows a technological scenario that describes the tasks to be performed to control the aggressive behaviour of a patient.

- Part of the tropos goal model. This model describes the behaviour of the system using a hierarchy of goals (from more general to more specific) and the tasks that must be performed for achieving the goals. The figure shows the decomposition of the goal [to reduce aggressive behaviour]. The model specifies that the system plans for achieving the goal [to reduce aggressive behaviour] are: [alert the nearest operator] and [change the environment for simulating relax] (e.g., put soft light and play music).

From the available requirements, we identified 4 behaviour patterns. Specifically, from the information shown in Figure 8.8, we identified the *Controlling aggressive behaviour* pattern that tries to relax a patient when s/he starts to behave aggressively. A detailed description of the behaviour patterns identified for the case study can be found in Appendix B.

Specify the Behaviour Patterns to Be Automated

We specified the identified behaviour patterns using the context and task models. To do this, we extended the context ontology with the type of users of the ACube project. The Tropos actor model and the

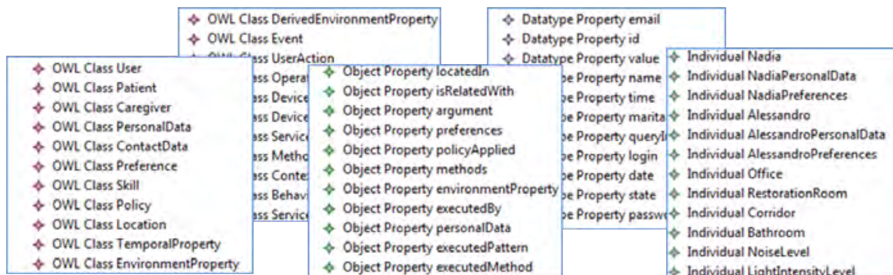

Figure 8.9: Overview of the context model created for the nursing home case study

set of *Personas* provided useful information for creating the needed hierarchy of users as subclasses of the *User* class in the context ontology. For instance, the actor model identified the roles *caregiver* and *patient*, while the *Personas* instrument identified more specific type of users: Carlo, who is a *patient with Alzheimer disease*, and Gianna, who is a *nurse* which is a type of caregiver. Thus, real users were specified in the user hierarchy as individuals of the class that better represented their characteristics. Figure 8.9 shows an overview of the context model of the case study.

Extended the context ontology, we could specify the four behaviour patterns identified for the ACube case study in the task model. Figure 8.10 shows the specification of these patterns using the task model. These patterns can be described as follows:

Controlling aggressive behaviour: This pattern is activated when a patient starts to behave aggressively. When this happens, the system captures the current context state (to be able to create the report at the end of the behaviour pattern execution). Then, the system alerts the nearest caregivers. To do this, it first searches for the caregivers that are nearest to the patient location and then sends them a message to warn that aggressive behaviour has been detected in the corresponding location. Then, the system changes the environment for simulating relax by putting soft lights and playing the preferred song of the patient that is behaving

aggressively. Five minutes later, if the patient is still behaving aggressively, the system warns the security officers. Afterwards, the system creates a report specifying: the context state in which the behaviour pattern was triggered, and the tasks carried out by the execution of the behaviour pattern. Finally, the report is sent to the involved staff so that they can validate it.

Avoiding Patient Escaping: This pattern is activated when it is detected that a patient is leaving the nursing home. When this happens, the system captures the current context state and then activates the emergency state and alerts the nearest caregivers. To do this, it first searches for the caregivers that are nearest to the patient location and then send them a message to warn that a patient is escaping. Then, the system starts to record the patient to see where s/he is going. Finally, a report about the incidence is created and sent to the involved staff so that they can validate it.

Dealing with a Fall: This pattern is activated when it is detected that a patient falls and none of the caregivers is around. When this happens, the system captures the current context state and then activates the emergency state and alerts the nearest caregivers. Finally, a report about the incidence is created and sent to the involved staff so that they can validate it.

Dealing with Health Emergencies : This pattern is activated when a health anomaly is detected in a patient. When this happens, the system captures the current context state and the nurses and the doctor of the patient are alerted. The external emergency light in the room is then switched on. When the situation is controlled, messages are sent to the involved personal staff to inform them about their next tasks. Finally, a report about the incidence is created in the health diary and sent to the involved staff so that they can validate it.

Since ACube is a user-intensive system, the behaviour patterns had to be specified for each user. To avoid this, we extended the

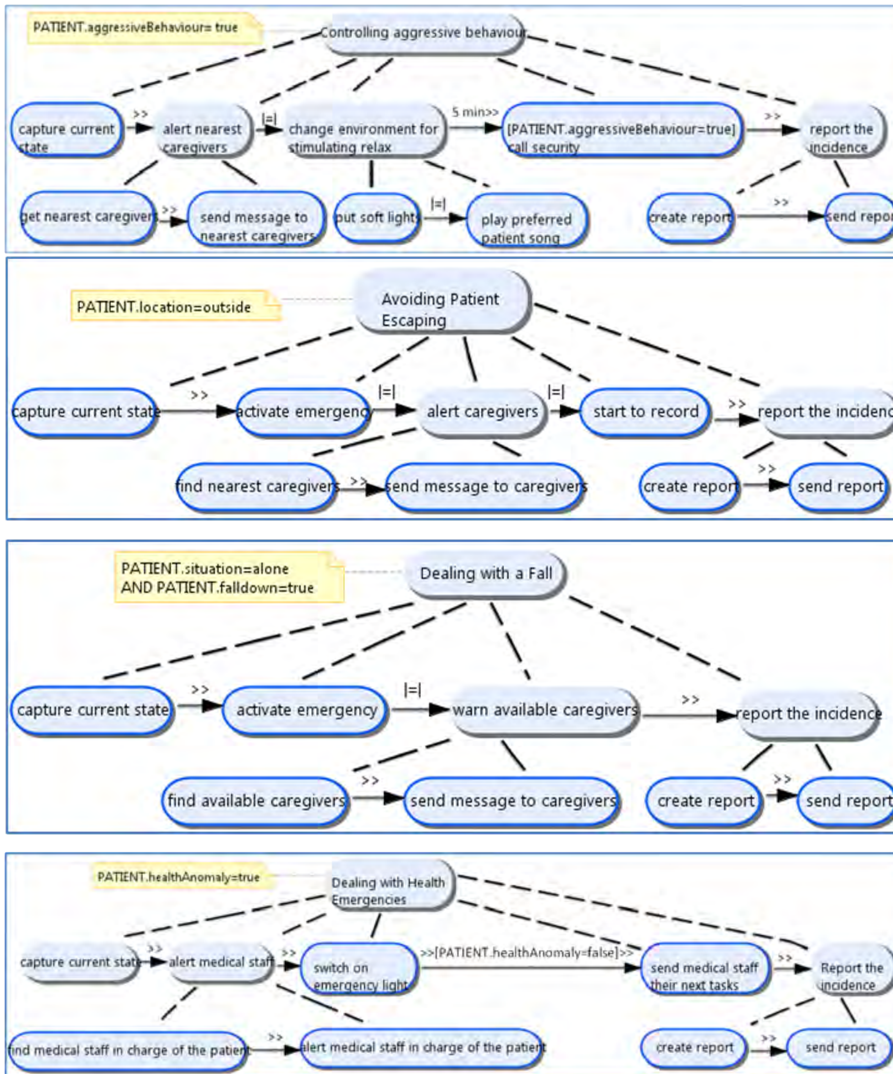

Figure 8.10: Specified behaviour patterns in the nursing home case study

checkCondition method in charge of checking the context conditions of the task model so that context ontology classes could be used in the

condition. Thus, if the name of a class appears in the condition, it is checked for all the individuals of the class. The condition is satisfied if one of the individuals fulfils the condition. For instance, the *Controlling aggressive behaviour* pattern has to be executed for every patient in which aggressive behaviour is detected. As shown in Figure 8.10, instead of specifying the same behaviour pattern for each patient, we specified the behaviour pattern once and used the *aggressiveBehaviour* context property of the *PATIENT* class in its context situation, indicating by using capital letters that it is an ontology class and not an individual. Thus, the context condition has to be checked for every individual of the *Patient* class and the pattern is activated if any patient fulfils the condition.

Testing the Provided Software Infrastructure

After the specification of the context and task models, we ran the system to check the automation and evolution of the specified behaviour patterns.

To support the functionality needed to execute the system tasks of the patterns, we developed the required services. Some of them, such as lighting or multimedia, were reused from the smart case studies. The other required services were implemented as simulated services in order to simulate the functionality of the needed devices because we did not have the real technology (t-shirt for monitoring patient health, user position detectors, etc.). Thus, we used a total of 17 different pervasive services. For more detail about these services, see Appendix B.

We then evaluated the feasibility of our approach. Using the running system, we passed the JUnit tests developed to check that the specified behaviour patterns were correctly automated as specified in the models (see 8.1.2). Since the automation of the behaviour patterns are triggered as a response to context changes, we caused these context changes by changing the state of the sensors using the simulator. We changed the state of the sensors simulating the scenarios of the requirement elicitation phase. For instance, to enable the *Controlling aggressive behaviour* pattern, we simulate that most of the patients were in the

dinning room and one of them start to behave aggressively. This makes the context situation of the behaviour pattern fulfil (see Figure 8.10).

In the same way, we simulated the rest of the scenarios of the case study and executed the prepared JUnit tests (see Section 8.1.2). For all of them, we checked that they were executed as specified in the models.

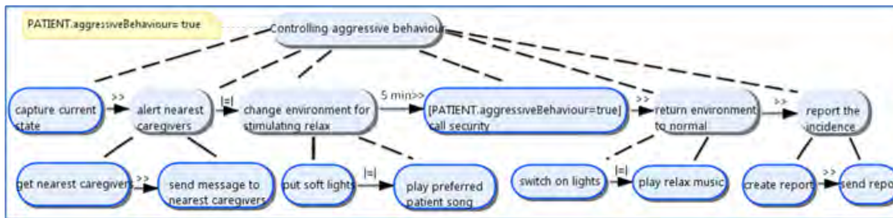

Figure 8.11: Example of a behaviour pattern evolution

We also performed some evolutions in the specified behaviour patterns. Figure 8.11 shows an example of these evolutions. It shows how the *Controlling aggressive behaviour* pattern has been extended to execute two tasks more, which return the environment to a normal state by switching lights on and turning on relax music. For each performed evolution, we applied again the JUnit tests checking that all the behaviour patterns were correctly executed.

8.2.3 Conclusions of the Nursing Home Case Study Validation

The development of the ACube Nursing Home case study has allowed us to answer the two research questions established at the beginning.

First of all, it is important to note that, the main goal of the ACube project was not specifically automating user behaviour patterns but to create an automated user intensive system. Thus, the requirements were not captured focusing on specifying behaviour patterns. However, in these requirements we could identify several behaviour patterns that could improve the medical and assistance staff tasks. Using these patterns, we have validated that the models proposed in this thesis

provide us with enough expressivity to describe them. However, since the developed case study is a user intensive system, we had to extend the notation of the context conditions specified in the task model to facilitate the specification of patterns that must be automated for several users (*Research Question 1*).

In addition, we developed the pervasive services that were needed to automate the patterns. We then ran the case study using these services, the specified models, and the provided software infrastructure. Testing the system in execution, we have validated that the behaviour patterns are automated as they are specified in the models (*Research Question 2*).

8.3 Scalability of Using Models at Runtime

Our software infrastructure manipulates models at runtime. This is subject to the same efficiency requirements as the rest of the system because the execution of model operations impacts overall system performance.

The developed case studies have required 12 behaviour patterns at most. Thus, we still have to validate whether our approach scales to large systems. To do this, we quantified the temporal cost of the operations of MUTate and OCean (the APIs that access models), for randomly generated large models.

We used our context model (in which the classes presented in Section 5.1 were defined) and an empty task model to be randomly populated by means of an iterative process. The context model was populated with 100 new individuals each iteration, while the task model was populated with one new pattern whose task structure formed a perfect binary tree, varying its depth and the width of the first level each iteration.

After each iteration, we tested all the model operations of MUTate and OCean 20 times and calculated the average temporal cost of each one. As an example, the operation over the context model with the highest temporal cost was the *checkCondition* operation, which took 7 milliseconds with 100 individuals and 10 milliseconds with 6000

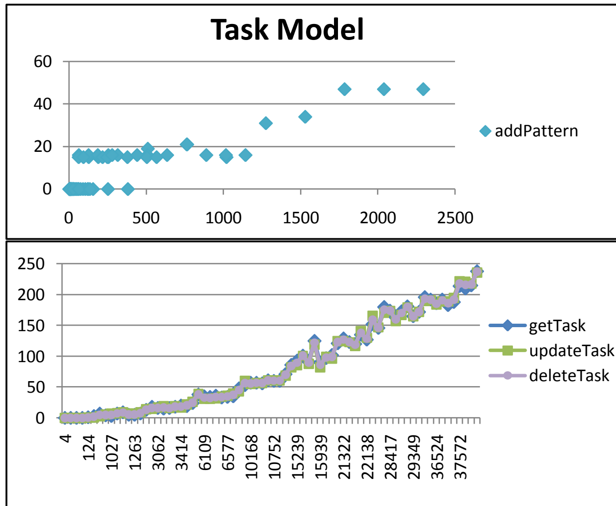

Figure 8.12: Temporal cost of task model operations

individuals. This is because this operation executes a SPARQL query, which determines the temporal cost of the operation. Figure 8.12 shows the temporal cost of the task model operations with the highest cost. At the top of the figure, we show the time required to add a behaviour pattern according to the number of tasks. This operation took less than 50 milliseconds to add a pattern of 2296 tasks. At the bottom, we show the *getTask*, *updateTask* and *deleteTask* operations. Their costs are very similar since all of them make the same query to obtain the corresponding task. Even with a model population of 45612 tasks, these model operations provided a fast response (<250 milliseconds). These results show that the response time is not drastically affected as the size of the models grows.

8.4 Conclusions

In this chapter, we have validated the proposal of this thesis for automating user behaviour patterns. To achieve this, we have evaluated

each one of the contributions of the work by using a case study based evaluation.

This evaluation allows us to conclude the following:

- The proposed task and context models provide us with enough expressivity to describe the behaviour patterns to be automated. However, as explained in this chapter, some aspects of the models can be improved to facilitate this specification. In addition, the proposed models are understandable enough for end-users to become into artefacts for discussion and for validating the behaviour that is going to be automated.

- The developed software infrastructure correctly automates the behaviour patterns as specified in the models by directly interpreting them at runtime and in a scalable way.

- The behaviour patterns can be correctly evolved by using OCean and MUTate and also using the graphical tool. In addition, this tool is usable enough to be used by end-users with computer knowledge.

Thus, we have shown that our approach is capable of automating the behaviour patterns that users want to be automated the way they want them to be.

Chapter 9

Conclusions

The present work has introduced a context-aware model-driven approach for confronting the challenge of automating user behaviour patterns. Confronting this challenge from a context-aware modelling perspective has allowed us to provide different and important contributions in the Pervasive Computing field that have resulted in relevant publications. In addition, the research line in which this work is aligned is by no means completed here. As it will be explained in this chapter, further work can complement and extend this thesis.

This last chapter introduces the conclusions of the work developed in this thesis. First, Section 9.1 presents the main contributions of our approach. Section 9.2 provides an overview of the publications that have emerged from this work. Finally, Section 9.3 outlines the ongoing and future work that can extend this line of research.

9.1 Contributions

The present work has introduced a novel approach from a modelling perspective that confronts the challenge of automating user behaviour patterns. To achieve this, the work provides the following contributions:

Modelling Language. We have proposed a context model and a context-adaptive task model that allow behaviour patterns to be specified in a context-adaptive way. By specifying behaviour patterns whose execution adapts to context, the intrusiveness that this execution may cause is considerably reduced. Also, to achieve a seamless automation of behaviour patterns, the models provide abstract concepts that facilitate the participation of end-users in the model specification. This favours that users' desires and demands are taken into account achieving the automation of the behaviour patterns that users want in the way they want it. Furthermore, the proposed models are specified in machine-processable languages and are precise enough to be used as executable models.

Behaviour Pattern Automation. Since the proposed models are machine-processable and precise enough, they can be directly used for automating the behaviour patterns (Pastor & Molina, 2007). Thus, we have designed and developed a model interpretation technique that interprets these models at runtime to execute the specified behaviour patterns. To achieve this, we have developed a software infrastructure that provides the following main components: 1) a set of mechanisms for managing the models at runtime; 2) a context monitor that is in charge of managing context so that the automation of behaviour patterns is performed unobtrusively; 3) MAtE, which is a model-based engine capable of executing the behaviour patterns as specified in the models.

Runtime Evolution of the Automated Behaviour Patterns. The developed model interpretation strategy facilitates the further evolution of the specified behaviour patterns to a large extent.

> This is because the patterns can be evolved by simply updating the models. Since they are interpreted at runtime, as soon as the models are changed, the changes are applied into the system. In addition, we have developed high-level mechanisms to support the evolution of the behaviour patterns by modifying the models at runtime. These mechanisms, which we refer to as OCean and MUTate, use the same high-level concepts for creating the models. Furthermore, we have developed a graphical tool that allows end-users to change the patterns by using user-friendly interfaces and without the need to stop the system or redeploy it. According to the changes described in these interfaces, the tool updates the task and context models by using OCean and MUTate.

In summary, we do believe that using a context-aware model-driven approach is a promising proposal to automate user behaviour patterns. This approach brings the following important benefits: user demands and desires are taken into account; tasks to be automated can be performed in a more pleasant manner for users and more efficiently regarding time and energy (since analysts define how the tasks must be performed with user participation); the cold-start problem is improved; and the runtime evolution of the automated tasks is greatly facilitated. Moreover, since the proposed models are not only used at design time but also at runtime, they can provide us with a rich semantic base for runtime decision-making. However, as the validation has revealed (see Chapter 8), some aspects of the models and the evolution tool have to be improved if our approach is to be used by end-users with a low level of computer knowledge or skills.

9.2 Publications

This approach has produced innovative and different contributions that have resulted in several relevant publications being discussed at different peer-review forums. In this section, we present the articles where this research has been published. In each one of the publications, the position of the name of the author of this thesis is used as an indicator

of the degree of contribution:

- **Estefanía Serral**, Pedro Valderas, Vicente Pelechano. *Automating Routine Tasks in AmI Systems by Using Models at Runtime.* International Joint Conference on Ambient Intelligence (AmI-10). In LNCS 6439, pp. 1-10. Málaga, Spain, November 10-12, 2010. ISBN: 978-3-642-16916-8
- **Estefanía Serral**, Francisca Pérez, Pedro Valderas, Vicente Pelechano. *An End-User Tool for Adapting Home Automation to User Behaviour at Runtime.* IV International Conference on Ubiquitous Computing and Ambient Intelligence (UCAmI 2010), pp 201-210. Valencia, SPAIN, September 7-10, 2010. ISBN: 978-84-92812-61-5
- **Estefanía Serral**, Pedro Valderas, Vicente Pelechano. *Improving the Cold-Start Problem in User Task Automation.* 19th International Conference on Information Systems Development (ISD 2010), pp. 648-659. Prague, Czech Republic, August 25-27, 2010. ISBN: 978-1-4419-9645-9
- **Estefanía Serral**, Pedro Valderas, Vicente Pelechano. *Supporting Runtime System Evolution to Adapt to User Behaviour.* The 22nd International Conference on Advanced Information Systems Engineering (CAiSE'10). In LNCS 6051, pp. 378-392. June 9-11, 2010. ISBN: 978-3-642-13093-9
- **Estefanía Serral**, Pedro Valderas, Vicente Pelechano. *Towards the Model Driven Development of Context-Aware Pervasive.* Special Issue of Pervasive and Mobile Computing (PMC) Journal on Context Modelling, Reasoning and Management. Vol. 6, no. 2, pp. 254-280. February 2010
- **Estefanía Serral**, Pedro Valderas, Vicente Pelechano. *A Model Driven Development Method for developing Context-Aware Pervasive Systems.* Ubiquitous Intelligence and Computing (UIC-08). In LNCS 5061/2008. pp. 662-676. Oslo, Norway, June 23-25, 2008. ISBN 978-3-540-69292-8

- **Estefanía Serral**, Pedro Valderas, Javier Muñoz, and Vicente Pelechano. *Towards a Model Driven Development of Context-aware Systems for AmI Environments.* International Conference on Ambient Intelligence Developments (AmI.d'07), pp. 114-124. Sophia Antipolis, French Riviera, September 17-19, 2007. ISBN: 978-2-287-78543-6

- **Estefanía Serral**, Carlos Cetina, Javier Muñoz, and Vicente Pelechano. *PervGT: Herramienta CASE para la Generación Automática de Sistemas Pervasivos.* XII Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2007), Zaragoza, Spain, September 11-14, 2007. ISBN: 978-84-9732-595-0

- Carlos Cetina, **Estefanía Serral**, Javier Muñoz, and Vicente Pelechano. *Tool Support for Model Driven Development of Pervasive Systems.* In 4th International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES 2007), pp. 33-44. In IEEE Computer Society. Los Alamitos, CA, USA, March 31, 2007. ISBN: 0-7695-2769-8

- Javier Muñoz, Vicente Pelechano and **Estefanía Serral**. *Aplicación del Desarrollo Dirigido por Modelos a los Sistemas Pervasivos: Un Caso de Estudio.* II Congreso IberoAmericano sobre Computación Ubicua (CICU 2006). pp: 171-178. Alcalá de Henares (Spain), June 7-9, 2006. ISBN/ISSN: 84-8138-703-7

- Javier Muñoz, **Estefanía Serral**, Carlos Cetina, and Vicente Pelechano. *Applying a Model-Driven Method to the Development of a Pervasive Meeting Room.* ERCIM News, April 2006 vol. 65, pp: 44-45. ISBN/ISSN: 0926-4981

- Javier Muñoz, Carlos Cetina, **Estefanía Serral**, and Vicente Pelechano. *Un Framework basado en OSGi para el Desarrollo de Sistemas Pervasivos.* 9º Workshop Iberoamericano de Ingenieria de Requisitos y Ambientes Software (IDEAS'06), pp: 257-270. La Plata, Argentina. Apr 24-28, 2006. ISBN/ISSN: 950-34-0360-X

- Javier Muñoz, Vicente Pelechano and **Estefanía Serral**. *Providing platforms for developing pervasive systems with MDA. An OSGi metamodel.* X Jornadas de Ingeniería de Software y Base de Datos (JISBD 2005), pp: 19-26. Granada, Spain. September 2005. ISBN/ISSN: 84-9732-434-X

9.2.1 Detail and Relevance of the publications

This section provides some information about the relevance of some of the journals and conferences where different aspects of this work have been published.

CAiSE and ISD. Both conferences are recognized as being one of the most important conferences in the area of information systems engineering. According to the CORE conference ranking, they are Tier-A conferences. In ISD, we published an overview of the approach focusing on the important benefits that it provides and the problems of other approaches that it attempts to solve.

The 22nd CAiSE conference had as special theme *Information Systems Evolution*. In this conference, we focus on the evolution of user behaviour patterns describing: the application of model interpretation and its benefits for evolving the patterns, OCean and MUTate, and the evolution tool. Moreover, this conference received 296 submissions, and only around 13% of the papers submissions were accepted.

PMC Journal. Pervasive and Mobile Computing is one of the most important journals in the Pervasive Computing Field. This journal is peer-reviewed and publishes high-quality scientific articles covering all aspects of pervasive computing and communications. Specifically, we published the specification and management of context information in the Special Issue *Context Modelling, Reasoning and Management* of this journal. According to the CORE conference ranking, this journal is a Tier-B journal.

UIC, AmI.d and AMI The Ubiquitous Intelligence and Computing

(UIC) conference and the International Joint Conference on Ambient Intelligence (AMI), which was previously known as the International Conference on Ambient Intelligence Developments (AmI.d), are international conferences that are very important in the area of Pervasive and Ubiquitous Computing. The initial development of the context ontology and context management was published in UIC and AmI.d. The proposed approach for automating behaviour patterns in which we explained the models and the designed software infrastructure was published in AMI. These conferences are peer-reviewed and, according to the CORE conference ranking, UIC is a Tier-B conference and AmI.d and AMI are Tier-C conferences.

International journals, conferences and workshops as well as national conferences. In addition to the above-mentioned conferences, the evolution tool was published in the international event UCAmI, which has special relevance for the Spanish research community focused on Ubiquitous computing and AMI systems.

Furthermore, this work has built upon previous research that had the goal of automatically building pervasive services. The research results and tools obtained resulted in three important international publications (MOMPES and CICU conferences and ERCIM News Journal) and two national conferences (JISBD and IDEAS). Essential knowledge for the development of the present work in pervasive computing (such as very relevant technologies and modelling techniques) was obtained from this collaboration. This has also helped to achieve diffusion for the work.

9.3 Future work

The research presented here is not a closed work and there are several interesting directions that can be taken to provide the proposal with a wider spectrum of application. The following list summarizes the research activities that are planned to continue this work.

9.3.1 Combination with Machine-learning Algorithms

As stated in Chapter 3, machine-learning approaches have done excellent work by providing prediction algorithms that infer user behaviour patterns from past user actions. These approaches have some important drawbacks such as the cold-start problem, loss of system control by users, users' desires and demands are not properly taken into account, etc. (see Chapter 3 for more details). However, we think that the integration of our approach with prediction algorithms could not only solve these drawbacks but also provide more automation in the evolution of behaviour patterns to adapt them to user behaviour.

In this work, we design behaviour patterns according to user requirements in a context model and a task model. The context model is continuously updated by the context monitor, which stores every context change and user action performed. When a change in context is detected, MAtE interprets the models and automates the corresponding behaviour patterns as specified.

Note that our work is made up of: (1) the models and the APIs to manage them at runtime; (2) MAtE and the context monitor, which use the models and the APIs to automate the user behaviour patterns specified in the models; and 3) the evolution tool, which provides end-users with graphical interfaces to allow them to update the automated behaviour patterns.

The integration of our work with prediction algorithms could use the task and context models as an initial knowledge base. This knowledge base, which is created at design time, would describe the behaviour that users need to be automated. This would avoid the cold-start problem because the system would start to automate actions from the system deployment. The access to this initial knowledge base would be managed by the MUTate and Ocean APIs. In addition, the knowledge base defined by models would provide prediction algorithms with initial data that can help them to make predictions. Furthermore, the models would allow this analysis to be performed using high-level concepts such as *task, location, user, context condition*, and so on, instead of analysing the raw data captured by sensors.

Once the system starts to run, the provided context monitor updates the context model according to context changes. The events that happen in the system are also considered as context. Thus, the context monitor not only updates the context properties whose value changes, but also the events produced that cause these changes. Prediction algorithms could use this information to automatically infer changes in the specified behaviour patterns or even infer new ones. In order to prevent users from losing control of the system and in order to take into account their desires, our evolution tool could periodically show these inferred changes to users instead of automatically applying them. Using the tool, end-users could make the changes that they consider opportune and add them to the system if they so desired. Thus, by combining our approach with machine-learning algorithms, the evolution of behaviour patterns could be performed in a more automatic way and users could keep control of the system.

9.3.2 Providing Adaptive User Interfaces

We plan to extend our evolution tool so that it provides interfaces that automatically adapt to each user. Thus, the evolution tool can better fit the needs of the end-users. Specifically, the tool interfaces could adapt to user preferences, skills, and knowledge of the system (Pribeanu *et al.*, 2001) changing how the functionalities are provided in the interfaces.

For instance, for users with little mathematics or computer skills, the context conditions definition can be difficult, as shown in Section 8.1.2. For these users, the corresponding interface could show them a list of already specified conditions, such as *when no one is at home*, *when the user enters the room*, *when it is cold*, etc. Thus, the user would only have to select the needed conditions from this list. However, this limits the conditions that can be formed because showing all the possible conditions would be unmanageable. Therefore, for users with mathematics or computer skills the current way of specifying the conditions would be better.

To achieve this adaptation, Feature Modelling techniques (Czarnecki *et al.*, 2004) can be used to describe adaptive interfaces (Gil *et al.*,

2010a). Feature Modelling is a technique to specify the variants of a system in terms of features (coarse-grained system functionality). The relevant aspects of each platform and the possibilities for their combinations are captured by means of the feature model. Features are hierarchically linked in a tree-like structure through variability relationships such as optional, mandatory, single-choice, and multiple-choice.

Besides describing the relevant aspects to the system, feature models have proven to be effective in hiding much of the complexity in the definition of the adaptation space (Cetina *et al.*, 2009). Thus, we can use Feature Models to describe the commonalities and differences between the tool interfaces in a declarative manner. In this way, the interfaces could be adapted to each user without explicitly defining how.

A designer should describe the possibilities for providing the functionalities in the interfaces according to context conditions (i.e., preferences, skills, etc.). In this way, interfaces are described using abstract aspects that can be mapped into different concrete representations depending on the device used. Thus, Feature models would allow us to compose the interface that is the most appropriate for each user without explicitly having to define it. This avoid duplicating efforts in the development.

9.3.3 Interactive and Iterative Tasks and Tasks with State

Although most of the behaviour patterns to be automated can be specified using the proposed task model, some behaviour patterns may require more expressiveness to be automated. Specifically, we consider that the extension of our approach would be of interest in supporting:

Interactive tasks: Tasks that require user attention may be needed in the automation of behaviour patterns because a complete automation may not always be desired. Users may want to know what is happening around them, or they may want the system to ask them to take some decision. For example, when the favourite

program of a user begins, the system should consider whether to start recording and/or informing the user depending on the context. If the system decides to inform the user first, it must choose the most adequate mechanism from all the ones available (i.e., sound, mobile vibration, a mobile text message, an email, a pop up, etc.).

Our approach can be extended to support interactive tasks. A new type of task should be created in the task metamodel, and the modelling tool should support the specification of this type of task. In addition, MAtE (the automation engine that interprets the task model) should be extended to manage this task adequately. To achieve this, when an interactive task must be executed, MAtE should create an interface that adapts its level of intrusiveness to the context of use (Gil *et al.*, 2010b). Thus, we could create a behaviour pattern that acts, for instance, as follows: 2 minutes before his/her favourite TV series starts, if the user is busy, the system could record the series (without informing the user); however, if the user is at home, the system could ask him/her whether the series should be recorded or the channel should be changed to show the series. This question may be done by voice if the user is alone on the sofa or by sending a mobile text message if there are other people in the room.

Iterative tasks: A behaviour pattern may require a set of tasks to be executed a certain number of times or while a certain condition is satisfied. Our approach can be easily extended to support this iteration. Every type of task might need to be iteratively executed; therefore, one way to support this would be to specialize the task class of the task model metamodel into an iterative task with two optional arguments: one argument to indicate how many times the task has to be executed and other argument to indicate the condition that must be satisfied to execute the task again. We would also need to extend the modelling tool to support the specification of iterative tasks. In addition, MAtE should be extended to manage this task adequately.

Tasks with state: In a behaviour pattern, it may be necessary to interrupt a task to start to execute another one. This is represented in the CTT temporal operators that need task interruption: $[<$, $|<$, $|||$, $|[]|$. To support these temporal operators, we need to consider tasks with state, i.e., we need to store the state of the tasks that are being executed. Moreover, we would have to consider the implications of interrupting a task in an ambient intelligence context, in which the tasks involved are usually tasks for controlling the state of the environment. According to these implications, we would have to give the operators the needed semantics and precision to be executed by model interpretation.

Thus, we would have to analyse what it means to interrupt each type of task in our approach. For instance, a system task cannot be interrupted because it is executed in an atomic way. However, this interruption may mean that the execution of another tasks is required to counteract the task that must be interrupted (record a film/stop recording, raise blinds/lower blinds, listen to music/stop music or turn down the volume, etc.). If the tasks are composite tasks, their interruption may mean that they must be stopped: no more of their system tasks must be executed (ever or until other task finishes). However, their interruption may also mean the counteracting of the already executed tasks.

Once the semantics of the relationships are precisely defined, we would need to extend MAtE to support them.

9.3.4 Facilitating the Routine Task Evolution by End-users

More facilities have to be provided to allow end-users to evolve the automated behaviour patterns over time. The designed evolution tool has shown to be effective in achieving this; however, some aspects of the interfaces have to be improved. Above all, more validations have to be developed in order to check that loops are not formed in the execution of the patterns and there are no inconsistencies with other patterns. To do this, the services provided by the pervasive system should provide

information about which operations perform contradictory tasks and also about the context properties that each service modifies in order to know if the execution of the services of a behaviour pattern can cause the execution of other patterns.

In addition, we plan to provide the evolution tool with simulation capacities so that the user can simulate the execution of the changed behaviour patterns to check beforehand if these patterns actually do what they want.

Figure 9.1: Snapshot of an iPhone interface for specifying a behaviour pattern

Furthermore, we plan to develop other types of interfaces that are easier and more accessible for every end-user, such as interfaces for mobile devices. Figure 9.1 shows examples of iPhone interfaces to allow users to create or modify a behaviour pattern.

Bibliography

Ajila, Samuel A., & Alam, Shahid. 2009. Using a Formal Language Constructs for Software Model Evolution. *Pages 390–395 of: Third IEEE International Conference on Semantic Computing.*

Annett, J., & Duncan, K. D. 1967. Task analysis and training design. *Occupational Psychology*, **41**, 211–221.

Ayed, Dhouha, Delanote, Didier, & Berbers, Yolande. 2007. Mdd approach for the development of context-aware applications. *Modeling and Using Context - 6th International and Interdisciplinary Conference, CONTEXT'07*, **Lecture Notes in Computer Science 4635**, 15–28.

Baldauf, M., Dustdar, S., & Rosenberg, F. 2007. A Survey on Context-Aware Systems. *International Journal of Ad Hoc and Ubiquitous Computing.*

Bardram, J. E. 2005. The Java context awareness framework (JCAF) - a service infrastructure and programming framework for context-aware applications. *Pages 98–115 of: Third International Conference on Pervasive Computing.*

Bennett, Keith, & Rajlich, Vaclav. 2000. Software Maintenance

and Evolution: A Roadmap. *Pages 75–87 of: 22nd International Conference on Software Engineering.*

Biegel, G., & Cahill, V. 2004. A framework for developing mobile, context-aware applications. *The 2nd IEEE Conference on Pervasive Computing and Communication (PerCom)*, 361–365. cortex, context model.

Blair, Gordon, Bencomo, Nelly, & France, Robert B. 2009. Modelsrun.time. *IEEE Computer*, **42**, 22–27.

Bohn, Jürgen, Coroama, Vlad, Langheinrich, Marc, Mattern, Friedemann, & Rohs, Michael. 2005. Social, Economic, and Ethical Implications of Ambient Intelligence and Ubiquitous Computing. *Pages 5–29 of:* Weber, W., Rabaey, J., & Aarts, E. (eds), *Ambient Intelligence.* Springer-Verlag.

Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., & Mylopoulos, J. 2004. Tropos: An agent-oriented software development methodology. *AAMAS*, 203–236.

Buckley, J., Mens, T., Zenger, M., Rashid, A., & Kniesel, G. 2003. Towards a taxonomy of software change. *Journal of Software Maintenance and Evolution: Research and Practice.*

Card, S.K, Moran, T.P., & Newell, A. 1983. *The Psychology of Human Computer Interaction.* Lawrence Erlbaum Associates. Cambrige Dictionaries Online.

Cetina, Carlos, Giner, Pau, Fons, Joan, & Pelechano, Vicente. 2009. Autonomic Computing Through Reuse of Variability Models at Runtime: The Case of Smart Homes. *IEEE Computer*, **42**(10), 37–43.

Chen, H., Finin, T., & Joshi, A. 2004. An ontology for context-aware pervasive computing environments. *Special Issue on Ontologies for Distributed Systems, Knowledge Engineering Review*, 197–207.

Chin, J.S.Y, V.Callaghan, & G.Clarke. 2008. A Programming-Byexample Approach to Customising Digital Homes. *IET International Conference Intelligent Environments.*

Cook, D. J., Youngblood, M., Heierman, III E. O., Gopalratnam, K., Rao, S., Litvin, A., & Khawaja, F. 2003. MavHome: An agent-based smart home. *Pages 521–524 of: In First IEEE International Conference on Pervasive Computing and Communications.*

Cook, Diane J., & Das, Sajal K. 2005. *Smart environments: technologies, protocols, and applications.*

Cooper, A., Reimann, R., & Cronin, D. 2007. *About face 3: the essentials of interaction design.* Wiley India Pvt. Ltd.

Crowley, J. L., Coutaz, J., Rey, G., & Reignier, P. 2002. Perceptual Components for Context Aware Computing. *International Conference on Ubiquitous Computing (UBICOMP 2002).*

Czarnecki, K., Helsen, S., & Eisenecker, U. 2004. Staged configuration using feature models. *Third Software Product Line Conference.*

David Wright, Elena Vildjiounaite, Ioannis Maghiros Michael Friedewald Michiel Verlinden Petteri Alahuhta Sabine Delaitre Serge Gutwirth Wim Schreurs, & Punie, Yves. 2005. The brave new world of ambient intelligence: A state-of-the-art review. *In: A report of the SWAMI consortium to the European Commission under contract 006507.*

Dey, Anind K. 2001. Understanding and Using Context. *Personal Ubiquitous Computing.*

Dey, Anind K., Hamid, Raffay, Beckmann, Chris, Li, Ian, & Hsu, Daniel. 2004. a CAPpella: Programming by Demonstration of Context-Aware Applications. *ACM Conference on Human Factors in Computing Systems (CHI 2004)*, 33–40.

Eclipse. 2011. www.eclipse.org.

Flyvbjerg, B. 2007. Five misunderstandings about case-study research. *Qualitative Research Practice: Concise Paperback Edition*, 390404.

France, Robert, & Rumpe, Bernhard. 2007. *Model-driven Development of Complex Software: A Research Roadmap.*

Gajos, Krzysztof. 2001. Rascal - a resource manager for multi agent systems in smart spaces. *CEEMAS 2001.*

Gajos, Krzysztof, Fox, Harold, & Shrobe, Howard. 2002. End User Empowerment in Human Centered Pervasive Computing. *International Conference on Pervasive Computing (Pervasive 2002).*

Galitz, O. Wilbert. 2002. *The Essential Guide to User Interface Design: An Introduction to GUI Design Principles and Techniques.* New York, NY, USA: John Wiley & Sons, Inc.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.

García-Herranz, Manuel, Haya, Pablo A., & Alamán, Xavier. 2010. Towards an ubiquitous end-user programming system for Smart Spaces. *Journal of Universal Computer Science (JUCS).*

Gemino, Andrew, & Wand, Yair. October 2003. Evaluating Modeling Techniques Based on Models of Learning. *Communications of the ACM*, **46**(10). modeling comprehensibility.

Gil, Míriam, Giner, Pau, & Pelechano, Vicente. 2010a. Designing context-aware mobile interactions. *In: 4th Symposium of Ubiquitous Computing and Ambient Intelligence 2010.*

Gil, Míriam, Giner, Pau, & Pelechano, Vicente. 2010b. Service obtrusiveness adaptation. *International Joint Conference on Ambient Intelligence (AmI-10)*, **LNCS 6439**, 11–20.

Google. 2007. *How To Design A Good API and Why it Matters.*

Gruber, T. R. 1993. A Translation Approach to Portable Ontology Specifications. *Pages 199–220 of: Knowledge Acquisition.*

Gu, T., Pung, H. K., & Zhang, D. Q. 2005. A service-oriented middleware for building context-aware services. *Journal of Network and Computer Applications*, **28(1)**, 1–18.

Guy, Marieke. 2009. *Report 2: API Good Practice Good practice for provision of and consuming APIs.* Tech. rept. UKOLN.

Haarslev, V., & Möller, R. 2003. *Racer: An OWL reasoning agent for the semantic web.*

Hagras, Hani, Callaghan, Victor, Colley, Martin, Clarke, Graham, Pounds-Cornish, Anthony, & Duman, Hakan. 2004. Creating an Ambient-Intelligence Environment Using Embedded Agents. *IEEE Intelligent Systems*, **19(6)**, 12–20.

Hartson, R., & Gray, P. 1992. Temporal Aspects of Tasks in the User Action Notation. *Human Computer Interaction*, **7**(1-45). UAN.

Heijden, Hans van der. 2003. Ubiquitous computing, user control, and user performance: conceptual model and preliminary experimental design. *A Research Agenda for Emerging Electronic Markets (RSEEM 2003).*

Henricksen, Karen, & Indulska, Jadwiga. 2004. A Software Engineering Framework for Context-Aware Pervasive Computing. *In: PerCom.*

Henricksen, Karen, & Indulska, Jadwiga. 2006. Developing context-aware pervasive computing applications: Models and approach. *Pervasive and Mobile Computing (PMC).*

Henricksen, Karen, Indulska, Jadwiga, & Rakotonirainy, Andry. 2006. Using context and preferences to implement self-adapting pervasive computing applications. *Sofware-Practice and Experience.*

Hervás, Ramón, Bravo, José, & Fontecha, Jesús. 2010. A Context Model based on Ontological Languages: a Proposal for Information Visualization. *J. UCS*, **16**(12), 1539–1555.

Hirschfeld, Robert, Kawamura, Katsuya, & Berndt, Hendrik. 2004. Dynamic Service Adaptation for Runtime System Extensions. *Software: Practice and Experience.*

Hofer, T., Schwinger, W., Pichler, M., Leonhartsberger, G., & Altmann, J. 2002. Context-awareness on mobile devices - the hydrogen approach. *The 36th Annual Hawaii International Conference on System Sciences*, 292–302.

Huang, Runcai, Cao, Qiying, Zhou, Jiliang, Sun, Daoqing, & Su, Qianmin. 2008. Context-Aware Active Task Discovery for Pervasive Computing. *In: International Conference on Computer Science and Software Engineering.*

Jochen Burkhardt, Thomas Schaeck, Horst Henn Stefan Hepper, & Rindtor, Klaus. 2002. *Pervasive Computing: Technology and Architecture of Mobile Internet Applications.*

John, B., & Kieras, D. 1996. The GOMS Family of Analysis Techniques: Comparison and Contrast. *ACM Transactions on Computer-Human Interaction*, **3**(4), 320–351.

Johnson, P. 1999. Tasks and situations: considerations for models and design principles in human computer interaction. *Pages 1199–1204 of: HCI International.*

Johnson, P., Markopoulos, M., & H, Johnson. 1992. Task Knowledge Structures: A Specification of user task models and interaction dialogues. *Proceedings of Interdisciplinary Workshop on Informatics and Psychology.*

Kulkarni, Ajay. 2002. *A reactive behavioral system for the Intelligent Room.*

Lauesen, S. 2003. Task Description as Functional Requirements. *IEEE Software*, **20**(2), 58–65.

Lee, A. S. 1989. A scientific methodology for MIS case studies. *MIS quarterly*, 3354.

Lenat, D. 1998. *The Dimensions of Context Space.* Tech. rept. Invited talk at the conference Context 99. Technical report, CYCorp.

Lewis, J. 1995. IBM Computer Usability Satisfaction Questionnaires: Psychometric Evaluation and Instructions for Use. *International Journal of Human-Computer Interaction*, **7 (1)**, 57–78.

Lieberman, H., & Selker, T. 2000. Out of Context: Computer Systems That Adapt To, and Learn From, Context. *IBM Systems Journal*, **39**, 617–631.

Lieberman, Henry, Paternò, Fabio, & Wulf, Volker. 2006. *End User Development*. Springer.

Lientz, B. P., & Swanson, E. B. 1980. *Software maintenance management: a study of the maintenance of computer application software in 487 data processing organizations.*

Limbourg, Quentin, & Vanderdonckt, Jean. 2004. Comparing Task Models for User Interface Design. *Pages 135–154 of:* Diaper, D., & Stanton, N.A. (eds), *The Handbook Of Task Analysis.*

Loy, Marc, Eckstein, Robert, Wood, Dave, Elliott, James, & Cole, Brian. 2002. *Java Swing*. O'Reilly, second edition.

March, Salvatore T., & Smith, Gerald F. 1995. Design and natural science research on information technology. *Decis. Support Syst.*, **15**(4), 251–266.

Mattern, Friedemann. 2001. The Vision and Technical Foundations of Ubiquitous Computing. *Upgrade*, **2**(5), 2–6.

Mattern, Friedemann. 2005. *Ubiquitous Computing: Scenarios from an informatised world*. Springer-Verlag. Pages 145–163.

M.B. Juric, B. Mathew, & Sarang, P. 2006. *Business Process Execution Language for Web Services: BPEL and BPEL4WS.*

Mellon, University Carnegie. 2009. *Alice: a programming environment for education.*

Mellor, S.J., & Balcer, M.J. 2002. *Executable UML: A Foundation for Model Driven Architecture.*

Mens, Tom. 2009. The ERCIM Working Group on Software Evolution: the Past and the Future. *In: IWPSE-Evolï¿½09.*

Mens, Tom, Wermelinger, Michel, Ducasse, Staephane, Demeyer, Serge, & Hirschfeld, Robert. 2005. *Challenges in Software Evolution: Report of the ChaSE 2005 workshop organised by the ERCIM Working Group on Software Evolution.* Tech. rept.

Miller, G. 1956. The magical number seven plus or minus two: some limits on our capacity for processing information. *Psychological Review 63*, 81–97.

Mitchell, K. 2002. *Supporting the Development of Mobile Context-Aware Computing.* Ph.D. thesis, Lancaster University.

Mozer, Michael C. 1998. The Neural Network House: An Environment that Adapts to its Inhabitants. *American Association for Artificial Intelligence Spring Symposium on Intelligent Environments*, 110–114.

Muñoz, Javier, Ruiz, Idoia, Pelechano, Vicente, & Cetina, Carlos. 2005. Un framework para la simulación de sistemas pervasivos. *Pages 181–190 of: UCAmI'05.*

Muñoz, Javier, Serral, Estefania, Cetina, Carlos, & Pelechano, Vicente. April 2006. Applying a Model-Driven Method to the Development of a Pervasive Meeting Room. *Pages 44–45 of: ERCIM News.*

Myers, Brad A., Pane, John F., & Ko, Andy. 2004. Natural programming languages and environments. *Commun. ACM*, **47**(9), 47–52.

Neal, David T., & Wood, Wendy. 2007. Automaticity in Situ: The Nature of Habit in Daily Life. *In: J. A. Bargh, P. Gollwitzer, and E. Morsella (Eds.), Psychology of action: Mechanisms of human action.*, vol. 2.

Nicholas Hanssens, Ajay Kulkarni, Rattapoom Tuchinda, & Horton, Tyler. 2002. Building agent-based intelligent workspaces. *ABA 2002.*

Nielsen, Jakob. 1993. *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Ormerod, T.C., & Shepherd, A. 2003. Using task analysis for information requirements specification: The SGT method. *In:* Diaper, D., & Stanton, N. (eds), *The Handbook of Task Analysis for Human-Computer Interaction*. London: Lawrence Erlbaum Associates.

OSGI. 2011. http://www.osgi.org.

Pastor, Oscar, & Molina, Juan Carlos. 2007. *Model-Driven Architecture in Practice: A Software Production Environment Based on Conceptual Modeling*. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

Paternò, Fabio. 2001. *Task Models in Interactive Software Systems*. Handbook of Software Engineering & Knowledge Engineering. World Scientific.

Paternò, Fabio. 2002. *ConcurTaskTrees: An Engineered Approach to Model-based Design of Interactive Systems.*

Paternò, Fabio. 2003. From Model-based to Natural Development. *HCI International*, 592–596.

Pérez, F., & Valderas, P. 2009. Allowing End-users to Actively Participate within the Elicitation of Pervasive System Requirements through Immediate Visualization. *In: REV'2009*.

Preuveneers, D., Bergh, J. V. den, Wagelaar, D., Georges, A., Rigole, P., Clerckx, T., Berbers, Y., Coninx, K., Jonckers, V., & Bosschere, K. D. 2004. Towards an extensible context ontology for ambient intelligence. *2nd European Symp. Ambient Intelligence*, **LNCS 3295**, 148–159.

Pribeanu, Costin, Limbourg, Quentin, & Vanderdonckt, Jean. 2001. Task Modelling for Context-Sensitive User Interfaces. *Pages 49–68 of: DSV-IS*. Springer-Verlag Berlin Heidelberg 2001.

Rashidi, Parisa, & Cook, Diane J. 2009. Keeping the Resident in the Loop: Adapting the Smart Home to the User. *IEEE Transactions on Systems, Man, and Cybernetics*, **39**.

Runeson, P., & Höst, M. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, **14**(2), 131–164. case study research.

Ryan, N. S., Pascoe, J., & Morse, D. R. 1998. *Enhanced Reality Fieldwork: the Contextaware Archaeological Assisstant.* Applications in Archaeology.

Satyanarayanan, M. 2001. Pervasive Computing: Vision and Challenges. *IEEE Personal Communications.*

Schilit, W. N., Adams, N. I., & Want, R. 1994. Context-aware Computing Applications. *In: Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications.*

Selic, Brian. 2003. The Pragmatics of Model-Driven Development. *IEEE Software.*

Serral, Estefanía, Valderas, Pedro, & Pelechano, Vicente. 2010. Towards the Model Driven Development of context-aware pervasive systems. *Special Issue on Context Modelling, Reasoning and Management of the Pervasive and Mobile Computing (PMC) Journal.*

Sheng, Q. Z., & Benatallah, B. 2005. ContextUML: a UML-based modelling language for model-driven development of context-aware web services. *Proceedings of the International Conference on Mobile Business (ICMB'05)*, 206–212.

Shepherd, A. 1993. An approach to information requirements specification for process control tasks. *Ergonomics*, **36**, 807–819.

Shepherd, A. 2001. *Hierarchical Task Analysis.* London: Taylor & Francis.

Silberschatz, A., Galvin, P.B., & Greg, G. 2004. *Operating System Concepts.*

Sirin, Evren, Parsia, Bijan, Grau, Bernardo Cuenca, Kalyanpur, Aditya, & Katz, Yarden. 2007. Pellet: A practical OWL-DL reasoner. *Journal of Web Semantics.*

Smith, Welty, & McGuinness. 2004. *OWL Web Ontology Language Guide.*

Sousa, JP, Poladian, V, Garlan, D, & Schmerl, B. 2006. Task-based Adaptation for Ubiquitous Computing. *IEEE Transactions on Systems, Man, and Cybernetics*, **36, 3**, 328–340.

SPARQL. 2010. SPARQL Query Language. http://www.w3.org/TR/rdf-sparql-query/.

Stalling, W. 2000. *Operating Systems: Internals and Design Principles.* Prentice Hall.

Truong, Khai N., & Abowd, Gregory D. 2004. INCA: A Software Infrastructure to Facilitate the Construction and Evolution of Ubiquitous Capture and Access Applications. *Pervasive Computing.*

Truong, Khai N., Huang, Elaine M., & Abowd, Gregory D. 2004. CAMP: A Magnetic Poetry Interface for End-User Programming of Capture Applications for the Home. *6th International Conference on Ubiquitous Computing (UbiComp)*, **3205 of Lecture Notes in Computer Science**, 143–160.

Uwe Hansmann, Lothar Merk, Martin S. Nicklous, & Stober, Thomas. 2001. *Pervasive Computing Handbook.*

Vaishnavi, V., & Kuechler, W. 2004 (January). *Design Research in Information Systems.* http://desrist.org/design-research-in-information-systems.

Valderas, Pedro. 2008. *A requirements engineering approach for the development of web applications.* Ph.D. thesis, Universidad Politécnica de Valencia.

Veer, G.C. van der, Lenting, B.F., & Bergevoet, B.A.J. 1996. GTA: GroupWare Task Analysis - Modelling Complexity. *Acta Psychologica 91*, 297–322.

Want, Roy, Hopper, Andy, Falcao, Veronica, & Gibbons, Jonathan. 1992. The Active Badge location system. *ACM Transactions on Information Systems*, **10**, 91–102.

Want, Roy, Schilit, Bill, Adams, Norman, Gold, Rich, Petersen, Karin, Goldberg, David, Ellis, John, & Weiser, Mark. 1995. An overview of the parctab ubiquitous computing experiment. *IEEE Personal Communications*, **2(6)**, 28–43.

Weiser, M. 1991. The Computer of the 21st Century. *Scientific American*, **265**, 66–75.

Welie, Martijn van, & Traetteberg, Hallvard. 2000. Interaction Patterns in User Interfaces. *Pages 13–16 of: Seventh Pattern Languages of Programs Conference.*

Westland, J. 2003. *Project Management Guidebook.*

Ye, Juan, Coyle, Lorcan, Dobson, Simon, & Nixon, Paddy. 2007. Ontology-based models in pervasive computing systems. *The Knowledge Engineering Review*, **22:4**, 315–347.

Youngblood, G. Michael, Cook, Diane J., & Holder, Lawrence B. 2005. Managing Adaptive Versatile Environments. *Pervasive and Mobile Computing.*

Appendix A

Software Infrastructure

This appendix provides detail on the implementation of the developed software infrastructure. The appendix is organized as follows: Section A.1 describes how the management mechanisms have been implemented. Section A.2 explains the implementation of the pervasive services. Section A.3 describes the context monitor in detail. Finally, Section A.4 presents detail information about the implementation of MAtE, the automation engine.

A.1 Model Management Mechanisms Implementation

In order to manage the context model and the task model (see Chapter 5) at runtime, we have defined and implemented two Java Application Programming Interfaces (APIs): OCean, which allows context models to be managed, and MUTate, which allows task models to be managed. OCean and MUTate provide the same vocabulary defined in the ontology and the task model metamodel, respectively. Thus, they

provide high-level abstraction mechanisms that facilitate the interaction with the models in order to achieve the automation and evolution of the specified user behaviour patterns.

OCean and MUTate are provided as java APIs to facilitate that they can be used by other software components in any software platform. The javadocs of these APIs can be downloaded from *http://www.pros.upv.es/art/*. To develop these APIs, we have applied the following best practices that have been recommended for developing APIs (Google, 2007; Guy, 2009):

- Planning: Before gathering data or developing something new, it is strongly recommended to check that there is not already a similar API available and if there are technologies that can provide us with a similar API.

- Keep it simple and easy to learn: The specifications must be simple and documented. Also it is recommended to avoid having too many fields and too many method calls. The API must offer simplicity, or options with simple or complex levels.

- Follow standards: It is advisable to follow standards where applicable. If possible it makes sense to use well-know standards from international authorities: IEEE, W3C, OAI or from successful and established companies.

- Use consistent naming structures: It is recommended to use consistent, self explanatory method names and parameter structures, explicit name for functions and follow naming conventions.

- Test the API: The API has to be checked and tested. It should be scalable, extendible and designed for updates.

Next, we explain the developed APIs and how we test their functionality.

A.1.1 Managing the Context Model: OCean

The context on which the behaviour patterns depend is specified in the context model as OWL individuals. Thus, in order to manage these individuals, we have implemented a set of Ontology-based Context model management mechanisms (OCean). OCean provides an implementation API that allows any individual of the context model to be created, obtained, modified, and deleted. For instance, OCean allow us to create a new user preference (e.g., idealTemperature), reading its value or modifying it when needed.

The API consists of a *Model* class that allows a context model to be opened and saved. Also, this Model class allows us to manage the individuals of the opened context model in a generic way. To achieve this, we have defined the *Instance* class that is composed of four elements: name of instance, name of the class which it is instance of, set of attributes and set of related instances. The names are defined as Strings; the set of attributes is defined as hashmap<String, String>, which stores the name and the value of the attribute, respectively; and the set of instances is also defined as a hashmap <String, Set<instance>>, which stores the name of the relationship and the instances related using this relationship, respectively. Using this Instance class, the *Model* class provides the following methods for managing the individuals in a generic way:

- addInstance: adds a new instance to the context model receiving the four elements which an Instance is composed of.
- deleteInstance: deletes the instance that is identified with the uri that receives as argument in String format.
- setAttribute: modifies the value of the corresponding attribute. To do this, the method receives as arguments the URI of the instance which the property belongs to, the URI of the corresponding property and the new value in String format.
- getProperty: returns the value of the property whose URI is received as argument in String format.

- setRelatedInstances: modifies the set of related instances. To do this, the method receives as arguments the URI of the instance which the relationship belongs to, the URI of the corresponding relationship and the set of instances that must be related.
- addRelatedInstance: add a new related instance. To do this, the method receives as arguments the URI of the instance which the relationship belongs to, the URI of the corresponding relationship and the new instance to be related.
- deleteRelatedInstance: delete a related instance. To do this, the method receives as arguments the URI of the instance which the relationship belongs to, the URI of the corresponding relationship and the URI of the related instance to be removed.

This *Model* class also provides facilities for querying the model using SPARQL (see 2.3), which is a graph-matching query language recommended by the W3C that allows queries to be built to search for certain individuals in the context model. Specifically, the class provides the methods checkCondition and selectElements that receive a SPARQL query in String format. The checkCondition method checks whether the query is fulfilled or not (it receives an ASK query), while the selectElements method obtains the corresponding elements that are selected with the SPARQL query (it is a SELECT query).

To implement this class, we have used Jena 2.4[1], the OWL API 2.1.1[2], and the Pellet reasoner 1.5.2. (see 2.3). Jena is a Java framework for building Semantic Web applications that provides a programmatic environment for OWL and SPARQL and includes a rule-based inference engine. We have used Jena to open the OWL model and save the performed changes in it. The OWL API is an open-source API that provides facilities for creating, examining and modifying an OWL ontology. We have used the OWL API to access to and modify the individuals of the context model. Pellet is an open-source OWL reasoner that provides reasoning services for OWL ontologies. Pellet facilitates

[1] http://jena.sourceforge.net/
[2] http://owlapi.sourceforge.net/

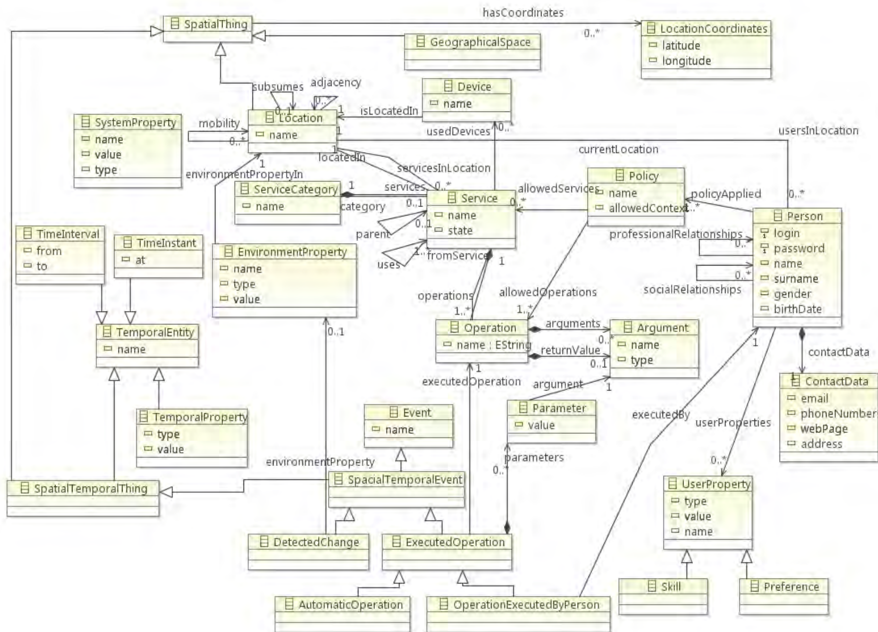

Figure A.1: Class diagram of the context ontology in ecore format

accessing to the information stored in the ontology and allows us to launch a SPARQL query against the context model using Jena.

In addition, OCean also provides a *Factory* class for creating new individuals in the context model and getting those that have already been created, and an implementation class (and its corresponding Java interface) for each one of the OWL classes defined in the context ontology so that the instances of the context model can be managed by using the same concepts defined in the context ontology. The class diagram of these classes is shown in Figure A.1. Each one of these Java classes provides:

- An attribute for each one of the properties and relationships of its OWL class; e.g., the User OWL class has DNI and preferences as attributes.

- Get, set and remove methods for each one of these attributes; e.g., getDNI or getPreferences.

- An add method for the attributes whose type is a List. This method allows an element to be directly added to the list; e.g., addPreferences method.

Several tools, such as Jastor[3], Jaob [4], Protégé or OWL2Java[5], have been developed for automatically generating a Java API (similar to the above described) from a given ontology for the handling of OWL instance data. After studying and trying these tools, we use the Jastor tool because it was the only one that generates most of the methods that we need, avoiding as much as possible concepts dependent of OWL technology in the API. From this generated code, we implemented the get method for the attributes whose type is a List, which was not generated. Also, the new methods generated by Jastor use the parameters *Model* and *Resource* from the Jena code; therefore, we add a *new* method that creates the new element only using the corresponding identifier to facilitate the use of the API and make it more independent of ontology technologies.

Figure A.2 shows an overview of the classes provided by OCean and a partial view of the source code of its *Person* class. Specifically, it shows the methods getSurname, which allows the surname of the person to be obtained, setSurname, which allows the surname of the person to be modified, and getSkills, which allows the skills of the person to be obtained.

A.1.2 Managing the Task Model: MUTate

In order to support the management of the task model, we have developed a set of Model-Based User Task management mechanisms (MUTate). For instance, MUTate allows searching for a behaviour

[3]http://jastor.sourceforge.net/
[4]http://wiki.yoshtec.com/jaob
[5]http://www.incunabulum.de/projects/it/owl2java

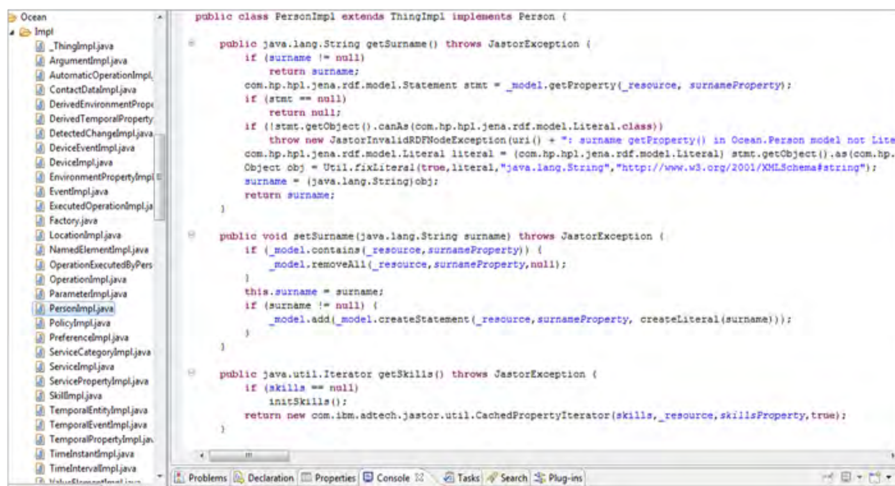

Figure A.2: Overview of the OCean API

pattern that have to be executed; adding new tasks to a pattern; creating a new pattern; etc. To do this, MUTate consists of a Java API that allows any elements of the specified task model (which are those specified in its metamodel, such as Behaviour Patterns, Tasks, Relationships between tasks, Conditions, etc.), to be created, obtained, modified, and deleted. Specifically, this API consists of a Factory class for creating new instances (of the classes defined in the task metamodel) in a task model, and a Java class for each one of the elements of the task model metamodel (see Figure 4). Each class provides:

1. An attribute for each one of the properties and relationships of the metamodel element that the class represents; e.g., the BehaviourPattern class has name and refinements as attributes.

2. Get, set and delete methods for each one of these attributes; e.g., getName.

3. An add method for the attributes whose type is a List. This method allows an element to be directly added to the list; e.g., addRefinement method.

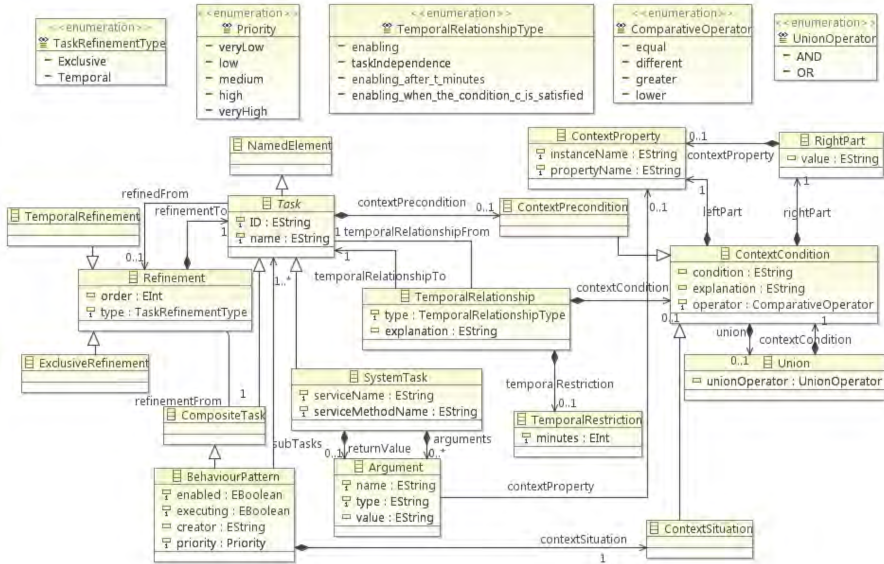

Figure A.3: Class diagram of the task model metamodel in ecore format

In order to implement this API, we have used the EMF, EMF Model Query (EMFMQ), and EMF Model Transaction (EMFMT) plugins of the Eclipse Platform (Eclipse, 2011), which provide us with many benefits for managing an XMI model at runtime.

From the metamodel of the task model in ecore shown in Figure A.3, we use EMF to generate a basic Java API for managing a task model that includes change notification, persistence support with default XMI serialization, and an efficient reflective API for manipulating EMF objects generically. This API provides the Factory class, as well as a Java interface and an implementation class for each one of the classes of the metamodel. These classes provide get and set methods to access and change the information of the instances specified in the model. We have extended these classes by implementing those methods explained in the last two points of the enumeration.

In addition, some of these Java classes represent context conditions, such as ContextSituation or ContextPrecondition. We have also implemented in these classes the checkCondition method to check whether the condition is fulfilled or not. This method interprets the logical expression of the condition and builds a query in SPARQL. Once the query has been built, the method uses the Model class of OCean to launch it against the context model.

EMFMQ facilitates the process of search and retrieval of model elements in a flexible, controlled and structured manner. To achieve this, this plugin allows us the construction and execution of queries in a SQL-fashion. The SELECT statement requires two clauses, a FROM and a WHERE. The former clause describes the source of model elements where SELECT can iterate in order to derive results. The latter clause describes the criteria for a model element that matches. Queries are first constructed with their query clauses and then executed against the model. We use the SELECT statements provided by this plugin to search for and get the instances of the model that need to be accessed or modified.

EMFMT provides us with mechanisms for making transactions, reading and writing models on multiple threads, and validating the semantic integrity of the modified model by detecting invalid changes. We have also extended the implementation classes provided by EMF with these mechanisms in order to: 1) add, modify, and delete a complete behaviour pattern as a unique transaction; 2) allow the reading and writing of the task model at the same time; and 3) semantically validate the changes in the task model.

Figure A.4 shows an overview of the classes provided by MUTate and a partial view of the source code of its implementation classes *TaskModel* and *BehaviourPattern*. Specifically, the figure shows the *getBehaviourPatternByContextSituation* method of the *TaskModel* class and the *getContextSitation* and *setContextSituation* of the *BehaviourPattern* class. The *getBehaviourPatternByContextSituation* method returns the behaviour pattern whose context situation is the same than the *BPContextSituation* argument value. To find the corresponding pattern, it searches for it by using a query statement

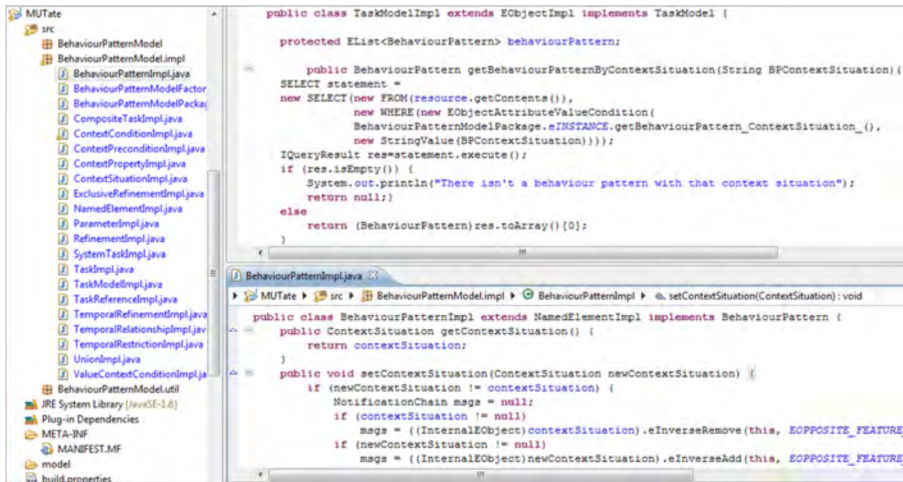

Figure A.4: Overview of the MUTate API

built with EMFMQ. The *getContextSitation* and *setContextSituation* methods obtains and modifies, respectively, the context situation of the behaviour pattern.

A.1.3 APIs' Testing

OCean and MUTate have to be tested to ensure that they provide the expected behaviour. Since the most part of MUTate and OCean have been generated by using code generation strategies already validated, we do not have to validate this code again. However, we have to validate the operations added to the generated code.

To test these operations, we used JUnit[6] tests. JUnit is an open-source unit testing framework for Java programs that provides functionality for writing and running unit test cases. A test case is a test class that describes test data, invokes the methods to test the methods of a class, and determines test results. To determine whether the method results are correct, *assertions* are provided. An assertion is a condition

[6]http://www.junit.org/

that should hold true after executing the method. JUnit provides the Equal, Not Equal, Same, Not Same, True and False assertions. After executing a test, the unit testing framework compares the actual value (the value returned after executing the code) with the expected value to determine the success or failure of the test.

Thus, we develop a JUnit test for each one of the operations that must be validated.

For instance, in order to check whether the *setProperty* method of the *Model* class of OCean was correct, we implemented a test method in the *ModelTest* class that call the setProperty to change the value of an attribute of a certain individual. The method then searches for the changed attribute by using the *getProperty*. This method returns the searched property value if it exists. Then, the method checks whether the returned value is equal to the new value set in the property by using an *Equal* assertion. Finally, we ran the implemented test method using the behaviour patterns and the context specified for the developed case studies (which will be explained in 8) as entry data. As shown in the figure, the time for running the test was 1,91 seconds, in which the time for opening the context model is included. We ran all implemented test classes in Eclipse. After correcting some errors, all the test methods were successful.

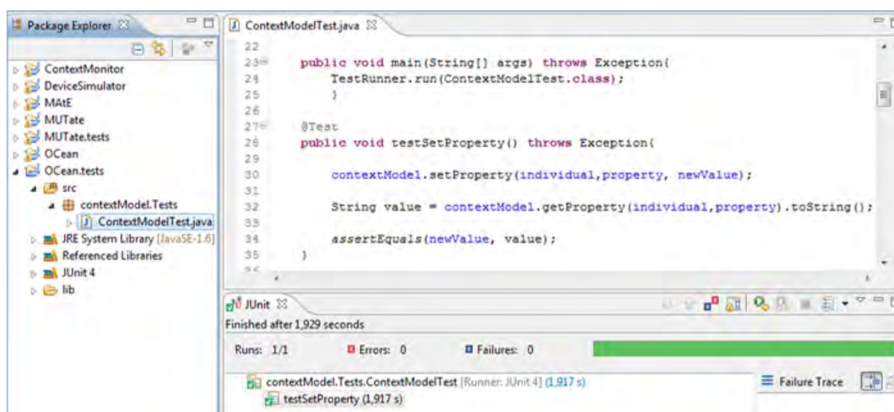

Figure A.5: JUnit test example

A.2 Pervasive Services

In order to automate the behaviour patterns specified in the models, our approach uses the pervasive services that every smart environment provides to control the devices of the environment. We consider a service to be an entity that provides a coherent set of functionality which is described in terms of atomic operations (or methods). These operations allows the system to change context and/or sense it.

Our approach attempts to be as independent of these service implementation as possible. However, so that these services can be used for the current implementation of our approach, they must fulfil the following requirements:

- They must be implemented using the OSGi/Java technology; specifically, each service has to be provided as an OSGi bundle. This technology is more and more used for developing pervasive services due to the numerous important benefits that it provides (see 2.4).

- Each service has to be registered as a service in the OSGi service registry by using a unique service id. This registry actually stores the interface that the service provides, which allows us to search a certain method of the service to be executed.

- Each service has to implement the OSGi Producer interface to be prepared for informing other services when it is used. For enabling uncoupled communication, OSGi allows communication channels, which are named wires in OSGi, to be established between bundles. An OSGI Wire is an enhanced implementation of the publish-subscribe pattern that is oriented to dynamic systems. In the OSGi framework, a Wire object connects a Producer service with a Consumer service, in such a way that they can communicate with each other via the wire. The Producer service may send updated values to the Consumer service by calling the *update* method of the wire (this method calls the *updated* method that must be implemented by the Consumer service).

Also, the Consumer service may request an updated value from the Producer service by calling the *poll* method of the wire (this method calls the *polled* method that must be implemented by the Producer service). In our approach, we need to automatically inform the consumers when a change is detected, therefore, we always use the wire *update* method.

- Each service must implement the following operations:
  - The *initializePersistentVariables* and *getPersistentVariables* operations, which initialize and returns, respectively, the values of the context properties that the service sense and that must be managed in the context model. This information is stored in a HashMap, in which the key is the individual and each key stores a hashmap with the name of the property as key and the property value as the key value.
  - The *hasChange* operation, which indicates whether the value of the context properties that the service manages has changed. If some properties has changed, this method returns them using the same structure used for storing the persistent variables.
  - The *consumersConnected* and *polled* operations, which are the operations that must be implemented for the OSGi *Producer* interface. The *consumersConnected* is used to know the OSGi bundles that are connected with the corresponding service and that must be warned when there some change detected by the service. The *polled* method implementation can be empty, since the communication in our approach is always produced from the producer to the consumer.
  - The *notifyConsumers* operation, which warns the context monitor if a change in some context property of the service is produced. This method must be called when a change can be produced (e.g., after the switch on operation of the Lighting service is executed). The *notifyConsumers* operation checks

```
/****
The method initializePersistentVariables must be implemented by every
pervasive service to establish which context properties are managed
by the service.
****/
protected void initializePersistentVariables() {
persistentVariables.put("Temperature",
                        new HashMap().put("value", getTemperature()));
persistentVariables.put("TemperatureLevel",
                        new HashMap().put("value", getTemperatureLevel()));
}
```

Figure A.6: An example of initializePersistentVariables operation that the pervasive services must implement

whether some change has been produced by calling the *hasChange* method. If some change has been produced, the operation calls the *consumersConnected* method to obtain all the wires that must be warned about the changes and executes the update method of these wires (this update method calls the updated method of the consumer service).

To facilitate the implementation of these methods, we provide the Service class that implements all of these methods except the initializePersistentVariables method that has to be implemented by each service. Thus, the pervasive services only have to inherit from this class and implement the initializePersistentVariables operation. Figure A.6 shows an example of the implementation of this operation for the service TemperatureMeasurement. This service senses the temperature of the environment and provides the exact temperature in Celsius degrees and its level according to the ranges high, medium, or low.

A.3 Context Monitor

The context monitor is in charge of updating the context model according to the context changes. These context changes are physically detected by the pervasive services (above explained) that control the

system devices. Thus, in order to capture context changes, the monitor is continuously monitoring the execution of the pervasive services. To do this, the context monitor implements the Consumer interface and creates a wire with each service (which implements the Producer interface). In a similar way than the Producer interface, the Consumer interface provides also two methods: producersConnected, which is used to know the OSGi bundles that are connected with the corresponding service and that warn it when they are used; updated, which is executed when some producer related with the service is used.

This OSGi implementation makes that when a change in a service is produced, the service notifies the context monitor about this change, since it is a consumer of the service. In this notification, the service sends to the context monitor a hashmap that contains the context variables whose value has changed. It can be seen like if this data was sending through the wire from the consumer to the producer.

When a change notification is produced, the updated method of the context monitor is called. This method, calls the updateContextModel method in order to reflect the changes in the context model by using OCean. In a change notification, the context monitor receives the name of each changed individual, the changed property and its new value. Using the OCean Model class, the monitor changes the values of the corresponding properties. The code implemented for reflecting the context changes in the context model is shown in Figure A.7.

Finally, once updated the context model, the updated method of the context monitor must inform MAtE about the context that has been updated. To do this, the context monitor and MAtE are also related by a wire. In this case, the context monitor plays the role of producer (implementing also the Producer interface), while MAtE plays the role of consumer (implementing the Consumer interface).

Thus, when the context monitor updates the context model, it notifies MAtE about the corresponding context change by calling the notifyConsummers method implemented in the context monitor. In this notification, the context monitor sends to MAtE a hashmap that contains the context variables whose value has changed.

```
/****
The updateContextModel method of the ContextMonitor is called
for updating the context model when a change in context is
detected by the pervasive services.
The method receives the detected context changes and updates
the context model by using the contextModel, which is an instance
of Model class provided by OCean.
****/
void updateContextModel(HashMap changes){
  String instance, property, value;
  Set changedInstances= changes.keySet();
  Iterator it_changedInstances=changedInstances.iterator();
  HashMap changedProperties;
  Iterator it_changedProperties;

  while (it_changedInstances.hasNext()){
    instance= (String)it_changedInstances.next();
    changedProperties= (HashMap) changes.get(instance);
    it_changedProperties=changedInstances.iterator();
    while (it_changedProperties.hasNext()){
      property=(String)it_changedProperties.next();
      value= (String)changedProperties.get(property);
      contextModel.setAttributeValue(instance, property, value);
  }
}
```

```
/****
The setAttribute method of the Model OCean class receives the ID of the
individual, the ID of its property and the new value that it must take,
and updates the value of the property.
****/
public  void setAttribute (String individualID, String propertyID,
                           String newValue)  {
OWLIndividual individual=
   factory.getOWLIndividual(URI.create(prefixURI+individualID));

OWLDataProperty dataProperty=
   factory.getOWLDataProperty(URI.create(prefixURI+ propertyID));
OWLDataType type= factory.getOWLDataType(URI.create(
   "http://www.w3.org/2001/XMLSchema#String"));
OWLConstant value=factory.getOWLTypedConstant(newValue, type);

OWLDataPropertyAssertionAxiom assertion_data= factory.
   getOWLDataPropertyAssertionAxiom(individual, dataProperty, value);
AddAxiom addAxiomData = new AddAxiom(ontology, assertion_data);

manager.applyChange(addAxiomData);
}
```

Figure A.7: Code for updating the context model

A.4 MAtE

MAtE is the engine in charge of automating the behaviour patterns in the opportune context by directly interpreting the models at runtime. To interpret the models, MAtE uses MUTate and OCean.

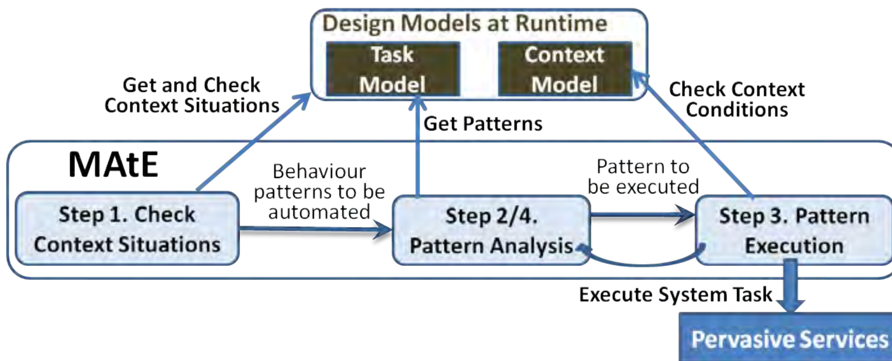

Figure A.8: MAtE process for automating the user behaviour patterns

As said above, to automate the behaviour patterns in the opportune context, MAtE is a consumer of the context monitor, since MAtE must check if any behaviour pattern must be executed when a change in context is produced.

Thus, MAtE implements the Consumer interface and creates a wire with the context monitor. This allows that when a context change is produced, the context monitor notifies MAtE about this change. When MAtE receives this notification, its *updated* method is executes. This method starts the automation process for automating those behaviour patterns whose context situation is fulfilled. This process consists of the following steps (which are summarized in Figure A.8):

1. Check the fulfilment of the context situations specified in the task model. Figure A.9 shows the code for performing this step. As shown in this code, to check the fulfilment of the context situations, MAtE first obtains the behaviour patterns specified in the task model by using the getBehaviourPattern method of the

TaskModel class of MUTate. MAtE then analyses the context situation of each behaviour pattern. To do this, MAtE first checks if the context situation is related to the performed context changes (i.e., if it contains some of the context properties that have been modified). If so, MAtE queries the context model to check whether this context situation is fulfilled by using the checkCondition method of the ContextSituation class. Finally, if a context situation is not fulfilled, MAtE removes its behaviour pattern form the list of behaviour patterns.

```
/****
Code for getting the behaviour patterns that must be executed (because
their context situation is now fulfilled).
****/
List behaviourPatternList= taskModel.getBehaviourPattern();
Iterator it_behaviourPatternList = behaviourPatternList.iterator();
BehaviourPattern BP;
ContextSituation CS;
while (it_behaviourPatternList.hasNext()){
    BP=(BehaviourPattern) it_behaviourPatternList.next();
    CS=(ContextSituation) BP.getContextSituation();
       if (CS.contains(changedContextProperties))
          if (!CS.checkCondition()) behaviourPatternList.remove(BP));
    }
}
```

Figure A.9: Code for carrying out the first step of MAtE by using MUTate

2. If there is some behaviour pattern to be executed, MAtE analyses their priorities and starts to execute the behaviour pattern with the highest priority.

3. MAtE executes the system tasks of the corresponding pattern according to its refinements, temporal relationships specified among its tasks and the up-to-date context information (stored in the context model) on which tasks and relationships depend:

   3.1. Its first refinement is obtained by using the getRefinementByOrder method (implemented in the CompositeTask class and inherited by the BehaviourPattern class).

      3.1.1. If it is an exclusive refinement, this means that only

one subtask of the composite task must be performed. This subtask is the first one (following the order of the refinements) that can be executed. Thus, MAtE gets the subtask of the refinement.

3.1.1.1. This task may have a context precondition, which means that the task must be only executed if this precondition is satisfied. Thus, if the task has a context precondition, MAtE checks it by using the checkCondition method of the ContextPrecondition class. If it is not satisfied, MAtE searches for the next refinement of the behaviour pattern and starts the step (3.1.1.1). However, if the task has not a precondition or it is satisfied, the task must be executed. If the task is a composite task, MAtE goes to the step 3.1 following a recursive process. If the task is a system task, it can be directly executed. To execute it, MAtE searches for the pervasive service related to the task to be carried out by using the OSGI capabilities, which allow services to be searched at runtime. Then, MAtE executes the service method associated to the task by using the Java reflection capacities, which allow us to execute a method by using its name, its arguments and its class name (i.e., the service name). The code for executing a system task is shown in Figure A.10.

3.1.2. If it is a temporal refinement, this means that all the subtasks of the composite task must be executed in the appropriate order. Thus, MAtE gets the subtask of the refinement:

3.1.2.1. MAtE executes the step 3.1.1.1.

3.1.2.2. MAtE checks if the task has a temporal relationship that links it to another task to be executed. If it has a temporal relationship, according to its temporal operator, MAtE executes the next task (NT) if its context precondition (if it has) is satisfied. If the

```
/****
The executeSystemTask method of MAtE executes the system task that
receives as argument by searching the pervasive service related to
the task and executing the corresponding service method.
****/
static Object executeSystemTask(SystemTask st){
Object res=null;
PervasiveService service= (PervasiveService)
serviceSearcher.getService(PervasiveService.class.getName(),
                              st.getServiceName());
try{
     Class serviceClass=service.getClass();
     Method[] metodos =serviceClass.getMethods();
     for (int i=0;i<metodos.length;i++){
          if (metodos[i].getName().equals(st.getServiceMethodName()))
              res=metodos[i].invoke(service,st.getParameters());
     }
}catch(Exception e){System.out.print("Excepcion: " + e.getMessage());}

return res;
}
```

Figure A.10: Code for executing a system task

temporal operator is >>[c]>>, MAtE will wait until the condition c is satisfied to execute NT, however, if the temporal operator is >> or |||, MAtE will not wait and directly will execute NT checking before its context precondition if had. To execute NT, the step 3.1.1.1 is carried out.

4. MAtE gets the behaviour pattern with the next highest priority. If its context situation is still satisfied, MAtE performs the step 3. If the situation is not satisfied, MAtE rules out the pattern and executes again the step 4. This step is performed until all the behaviour patterns obtained in the step 1 have been analysed.

Appendix B

Case Study Requirements

B.1 Smart Home Requirements

The objective of the smart home case studies was to improve users' lives and saving energy resources by automating daily tasks of the users. In order to capture the automation requirements for these case studies, we design an interview for interviewing the end-users of the case studies. From these interviews, we identified the routines that users want to be automated. Finally, we detected the services that were required for automating the identified routines.

B.1.1 An Interview for Identifying the Behaviour Patterns

In order to identify the behaviour patterns that users want to be automated, we designed a semi-structured interview. A semi-structured interview is composed of planned questions, but they have not to be necessarily asked in the same order as they are listed. Thus,

we could decide in which order the different questions should be handled according to the development of the conversation in the interview. Also, we could use the list of questions to be certain that all questions were handled. Additionally, semi-structured interviews allow for improvisation and exploration of the studied objects; therefore, we could improvise more questions if needed.

To prepare the interview, we followed the advices published in (Runeson & Höst, 2009). We first presented the objectives of the interview and the case study, and explained how the data from the interview will be used. Then, a set of introductory questions that are relatively simple to answer were asked to the subject, such as his/her name, what he or she does, etc. After the introductory questions, we performed the main interview questions following a time-glass model, i.e., beginning with open questions, moving towards more specific ones focusing on the routines performed by the subject and opens up again towards the end of the interview. Although the interview was individual, when the subjects of the case study were a family or a couple, all the members were present to avoid hitches. The questions that we prepared for the interview were the following:

- Do you perform routine tasks?
- Would you like that they were automated as you want?
- Which routines/habits do you perform on a working day that you would like to be automated? What do you usually do on a working Monday, Tuesday, etc.?
- Do you perform routine tasks in weekends? Which routines do you perform?
- Do you think that the automation of these routines would improve your quality of life? How?
- Do you think that the automation of these routines would reduce energy and water consumption?

Using this interview, we interviewed the participants recording the conversation as recommended(Runeson & Höst, 2009). In the answers of the interview questions we observed that users described the context situation that triggers the routines and the context conditions of the tasks by using the word *when*. Also, they naturally describe the context conditions by answering to the question *in which circumstances*. In addition, we noted that, in general, married respondents had family focused responses while single people, even not living alone, described their tasks in an individual manner.

After they described the routines that they performed, we also proposed them some behaviour patterns that may be very useful, such as presence simulation for getting away thieves when users are not at home, tasks to preserve security, control of blinds to save energy, watering the garden by saving energy, etc. We found that most of the interviewed users would like to have automated many of the proposed behaviour patterns; however, everyone had its own small variations.

Regarding the last general questions of the interview, all the users commented that they would love the routines that they wanted were automated. All of them said that this automation would improve their quality of life. Some of the reasons that they argued were: 1) this automation would make me avoid worrying of the tasks to be done; 2) this automation would let me more free time; 3) this automation would make my life more comfortable. Also, all of the interviewed subjects thought that the automation of their routines would reduce their resource consumption.

B.1.2 The Identified Behaviour Patterns

We analysed the information obtained from the interviews and identified the behaviour patterns that could be useful for the users. We identified from 6 to 12 behaviour patterns to be automated in each case study, with a total of 97 behaviour patterns. In essence, from these behaviour patterns we detected 15 that were different, i.e., that had different goals. The rest of behaviour patterns were variations of them. For instance, most users wanted automatically room lighting taking into

account outside light intensity and user presence. If the user slept alone, he or she wanted the room was always illuminated when s/he wakes up, however, if users were a couple or have babies, they usually wanted the light in the bedroom was not switched on when there was someone sleeping. In contrast, another couple wanted the lights were not switched off while her daughter was playing in the house.

The 15 different behaviour patterns that were identified can be described as follows:

Presence simulation: It simulates that users are at home when they are going to stay out (e.g., holidays, weekends, etc). To perform this simulation, the pattern executes the tasks for controlling lighting, TV and radio that users usually perform when they are at home.

Home Security: When an intruder, a gas leak, a water leak, or a fire is detected, all lights in the house blink to alert any occupants of the house. All audio and video components are switched off to avoid distractions,. The system could also call the home owners on their mobile phone to alert them, or call the fire department or alarm monitoring company.

Lighting Control: Lights and blinds are controlled to light the room when needed and to save energy when possible.

Storm Security: Blinds, awnings, windows and sprinklers are controlled to avoid water gets into the house and windows are got dirty when it starts raining.

Waking Up: This pattern executes the tasks that users want for waking them up in a more comfortable way.

Leaving Home: This pattern executes the tasks to be done when users leave home (e.g., switching off lights, controlling heating and air conditioning for saving energy, etc.).

Going to Bed: This pattern executes the tasks to be done when users go to bed (e.g., switching off lights, controlling heating and air

conditioning for saving energy, etc.).

Getting a Comfortable Temperature: Air and heating conditioner and windows are automatically controlled to achieve the best temperature in each room according to: user presence, user preferences, inside temperature and outside temperature; saving energy as much as possible.

At Night: When users are at home, blinds are lowered to preserve privacy.

Watering the Garden: The garden is watered when recommended without bothering users and until it achieves the appropriate humidity.

Stop Watering: If someone goes out to the garden, the sprinklers are switched off. When no presence is detected in garden, the sprinklers are switched on again if needed.

Faucets Control: The taps are opened and closed when needed to save water.

Watching a Movie: Home cinema is prepared, the blinds are lowered in the living room and its light intensity is lowered.

Door Control: Doors are opened when users are detected to go through them. If there are little children in home, certain doors, like the kitchen door, will not be opened for children security.

Answering a Call: This pattern turns on volume of the sound devices that are producing sound when the telephone rings. When the phone call finishes, the pattern puts again the normal state. It can be configured depending on the person that calls, even it could reject phone calls of numbers that users had previously predefined.

B.1.3 Required Services

In order to support the automation of the identified behaviour patterns, the services shown in Figure B.1 were required.

Figure B.1: Services required for the smart home case studies

B.2 Nursing Home Requirements

The ACube[1] project has been designed to be deployed in a nursing home in Trento. ACube is a large research project funded by the local government of the Autonomous Province of Trento in Italy with the aim of designing a highly technological smart environment to be deployed in nursing homes as a support to medical and assistance staff. The system is based on a network of sensors distributed in the environment or embedded in users' clothes.

In this appendix, we explain in detail the artefacts obtained from the requirement elicitation process, the behaviour patterns identified using these artefacts, and the services required for supporting these patterns.

B.2.1 ACube Requirement Elicitation Artefacts

In this section we explain the requirement elicitation artefacts that we had at our disposal: a tropos model, a set of *personas* and a set of scenarios.

Tropos Model

The **Tropos Model** is used for modelling the set of domain entities when the system is not yet existing. It includes a bird-eye view over the domain in which actors and roles are specified together with their responsibilities and delegations. This view provides an intuition of which interactions occur in the environment. Subsequently each actor is exploited in a goal model, in order to provide details about human behaviour, highlighting the rationale by relating each activity to institutional motivations. The Tropos model designed for the ACube case study is shown in Figure B.2.

Personas

The *personas* identified for this case study were the following:

[1] http://acube.fbk.eu/

Figure B.2: Tropos model of the ACube case study.

Name: Sabrina

Age: 40 years old

Description: She has been working as a caregiver in the nursing home for 5 years

Goal: To assist guests in all their daily activities

Problems: She likes the social side of her work. She complaints to have not time for establishing good relationships and to know guests The night turn is the most difficult since she is alone for 8 hours with 36 guests She is not comfortable with technology and thinks the computer is too difficult to use. Wishes: She would like to have more time for improve the knowledge of her guests. She would work in a more friendly structure, in which guests are free to move in and out.

Name: Gianna

Age: 38 years old

Description: She has been working as a nurse for 2 years in the nursing home

Goal: To provide sanitary assistance and administer therapies to guests

Problems: She complains the lack of time to carry out all duties. The bureaucracy is too heavy. She would use new technologies.

Name: Maria

Age: 78 years old

Description: She has been in the nursing home for three months and is affected by senile dementia. She has problems with memory and disorientation. She is not under specific monitoring because she have never tried to escape. She can walk though the recent assessment made by the physiotherapist gives some balance

problems. She moves by the sustain of the handrails or by using the stick.

Wishes: Maria wants to remain independent even if she is in the nursing home. She would like to be able to move in the centre without the help of operators, see her family more often and do more recreational activities

Name: Carlo

Age: 93 years old

Description: Carlo has been in the nursing home for 1 year. He is suffering from Alzheimer's disease, memory deficits, disorientation in time and space and behavioural problems. He once tried to escape, so operators should give special attention to his movements. He has been aggressive in past; this crisis had been handled promptly by the operators who must appease him by distracting him away from other guests and by means of its interests (e.g., singing).

Wishes: Carlo smokes and would like to stay outside at fresh air. He often complains because he does not like to stay in nursing homes.

Name: Piera

Age: 90 years old

Description: Piera has been in the nursing home for 6 years. She has mobility problems which prevent her from walking. She also has health issues (blood glucose and cardiac problems that require constant monitoring, trauma to the femur). In addition, she is impaired in cognitive deficits: memory and depression. She needs for constant assistance.

Wishes: Piera has problems with depression. The situation leads her to loose motivations. She would like to have a more human relationship with operators, doctors and nurses.

Scenarios

The following scenarios have been identified in the ACube project:

Scenario 1: Fall monitoring and prevention. Maria is leaving the restoration room, and the sensor on the door sends a signal to Sabrina's PDA that alerts with a vibration. Sabrina knows that a vibration means Maria is moving out from the room, but she cannot follow her in that moment because she must oversee the room. Whether Maria leaves the room with other guests or with a caregiver, the alarm would not be sent. Maria is going upstairs in order to reach her private room but when she is on the staircase, she falls. The camera identifies the event and sends warning signals to caregivers' PDA. Sabrina's PDA displays an unknown person is fallen down in the staircase between second and third floors. The nurse, Gianna, receives this signal and she is available to go, so she notifies (by PDA) that she is taking the event in account. Also Sabrina decides to go, she imagines that Maria is fallen, so she sends a message to other caregivers that restoration room is currently not overseen. Renato (that is about to finish his turn) receives the message and suddenly goes to the restoration room where guests are alone. Sabrina reaches Maria and soothe her. Maria is active and she talks and reasons perfectly, she is afraid but she is not in pain for the hit. Gianna rapidly understands that all is OK and she press the orange button on her PDA (emergency is off). Maria is helped to stand and to return in her room. Sabrina comes back to other guests thus Renato is free to go home. At the end of their turn, Sabrina and Gianna have to write the report for the next turn colleagues. They turn on their computer and find an automatic report with all data relative to the event. Cameras, audio and RFID sensors have collaborated to collect data and to compile the report.

Scenario 2: Escape monitoring and prevention. Carlo is in the garden and follows some visitors going through the gate with the intention to run away. Carlo's bracelet sends Carlo's position

to the system. The alert signal comes to Sabrina's PDA who reads "Carlo is leaving the institute", thus she decides to go. She communicates by using the PDA that is is taking in account the emergency. Other caregivers receive only a warning message. The camera near the gate activates and: 1) tries to follow Carlo's path 2) automatically locks the gate to prevent the escape Whether Carlo goes through the gate a second RFID sensor sends a message to Sabrina (who takes in account the event) alerting that the emergency is now serious. The camera records all the activities thus to allow caregivers to see what happened. All the events are collected in order to write the report at the end of the turn.

system knows that in last days Carlo is quite, likes to stay in the garden and smokes less.

Scenario 3: Aggressive behaviour. Sabrina has just started her turn. She is alone in the great restoration room where there is a group of Alzheimer people (8-10). Today it is noise and Sabrina can not oversee everyone. It is summer and a few social workers are in the institute. Piera begins to disquiet and her behaviour becomes aggressive. A camera in the room identifies Piera's state and the system switch on some soft lights around Piera, and plays her preferred song. In the meanwhile the system alerts the nearest caregiver Sabrina about the trouble who decide whether to go, to call help or to ignore it.

At the end of the turn, Sabrina is in her office and validates the automatic report that describes what happened: Piera's behaviour, and the action activated (lights and music) and Piera's response. The report also contains that the room was full and maybe this is the cause of Piera's stress. The system learns something new.

Scenario 4: Night monitoring. Two caretakers are working in couple during the night to oversee and support guests during the night. They must move each guest every three hours. In the institute only a doctor is present. They are in Piera's room and all is OK, thus they continue their work. Suddenly Piera is suffering

a heart attack; the t-shirt identifies the event. The PDA soon alerts Sabrina and Manuela (and the nurse and the doctor) with a vibration and the text: heart attack on room... The external light in the room silently switches on to drive caregivers to the right room. This signal can be switched off manually. In the meanwhile Maria coughs and microphones identify the event. In this case only Sabrina and Manuela receive the warning because Maria's T-shirt estimates a regular breath and ECG (it is not a health emergency). When the nurse and the doctor are in the room, the caretakers can leave the room and continue their work. They can ask the system the last guest they have supported so to avoid to forget someone. The system replies that room 123 is completed but in 124 Ugo must be moved. The health emergency is automatically reported in the health diary.

B.2.2 The Identified Behaviour Patterns

We analysed the requirement artefacts that we had available and identified 4 behaviour patterns that could be useful for supporting the tasks of medical and assistant staff. These patterns can be described as follows:

Controlling Aggressive Behaviour: When a patient starts to behave aggressively, the system alerts the nearest caregivers. Then, the system puts soft lights and plays the preferred song of the patient that is behaving aggressively. Five minutes later, if the patient is still behaving aggressively, the system warns the security officers. Finally, a report is created and sent to the involved staff.

Avoiding Patient Escaping: If it is detected that a patient is leaving the nursing home, the system activates the emergency state, alerts the nearest caregivers and starts to record the patient. Finally, a report is created and sent to the involved staff.

Dealing with a Fall: If it is detected that a patient falls and no one of the caregivers is around, the system activates the emergency

state and alerts the nearest caregivers. Finally, a report is created and sent to the involved staff.

Dealing with Health Emergencies: When a health anomaly is detected in a patient, the nurses and the doctor of the patient are alerted. The external emergency light in the room is then switched on. When the doctor and the nurses arrive to the room, the emergency light is switched off. If the situation is controlled, messages are sent to the involved personal staff to inform them about their next tasks. The health emergency is automatically reported in the health diary.

From Requirement Models to Executable Models

In order to automate the process of obtaining the context and task models from the requirement artefacts (*personas*, scenarios and Tropos models), we proposed a methodology in collaboration with the research group in charge of the ACube project in the Fondazione Bruno Kessler. The steps of the proposed methodology are described as follows:

Step 1: Detect the behaviour patterns to be automated. The step consists in identifying the behaviour patterns that can be automated by the system. To identify them, the Tropos goal model is used. A one-to-one relationship is identified between goals delegated to the system and behaviour patterns (e.g. the goal [to reduce aggressive behaviour] could be transformed into a behaviour pattern named *Controlling aggressive behaviour*).

Step 2: Model the task hierarchy of each behaviour pattern. Each behaviour pattern is specified using a task hierarchy, from more general to more specific tasks. This hierarchy is obtained from the task decomposition of the corresponding goal in the tropos goal model. This can is completed with the information provided by the technological scenarios: the action verbs whose subject is the system represent tasks to be automated (e.g., *the system plays his (Carlo) preferred song*). An example of task hierarchy obtained following this guideline is shown in Figure B.3.

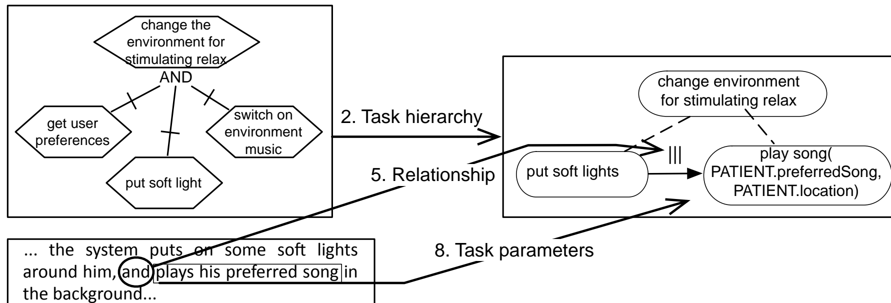

Figure B.3: Transformation performed following the provided guidelines

Step 3: Specify users. The users involved in the tasks to be automated are identified. The Tropos actor model and Personas provide useful information for creating a hierarchy of users, which has to be specified in the ontology as subclasses of the *User* class. For instance, the actor model identifies the roles *caregiver* and *patient*, while the Personas instrument identifies more specific type of users: Carlo, who is a *patient with Alzheimer disease*, and Gianna, who is a *nurse* which is a type of caregiver. Real users will be specified in the hierarchy as individuals of the class that better represent their characteristics.

Step 4: Specify context. Tasks to be automated usually depend on context information. This context information appear in the scenarios as adjectives (e.g. *noisy*), locations (e.g., *restoration room*), temporal aspects (e.g., *season*), etc. Also, the motivations of the goals specified in the Tropos goal model can be used for detected needed context information (e.g. *aggressive behaviour*). The identified context properties must be specified in the context model as individuals of the corresponding ontology classes (e.g., noisy should be an instance of the EnvironmentProperty class).

Step 5: Specify temporal relationships. If a behaviour pattern, or a composite task, has been refined by temporal refinements, its subtasks have to be related between them by using temporal relationships that rigorously specify the execution order of these subtasks. Scenarios can help to define these relationships. For instance,

as shown in Figure B.3, in the scenario it is explained that the systems puts soft lights and plays Carlo's preferred song, meanwhile, the system alerts the caregivers. From this information, we can deduce that the order of execution of these tasks is not important, then, the $| = |$ relationship must be used.

Step 6: Specify the context situation. Each behaviour pattern has to be related with a context situation whose fulfilment activates the execution of the pattern. The meaning of the goal to be achieved as well as the technical scenarios can help to define these context situations. For instance, to achieve the goal [to reduce aggressive behaviour] the identified controlling aggressive behaviour pattern must be activated when an aggressive behaviour is detected, as also is explained in the scenario 3 (*...his behaviour becomes aggressive. A camera in the room identifies it and the system ...*).

Step 7: Specify context dependencies. The specified tasks may have to be executed only when some context conditions are satisfied. Thus, these conditions are specified as task preconditions by using the context properties identified in Step 3. For instance, the *call security* task will be executed if the user continues behaving aggressively after executing the previous tasks, then, the context precondition *aggressiveBehaviour=true* must be added to this task.

Step 8: Specify task parameters. If a system task need parameters to be performed. To detect these parameters, resources in goal models and technological scenarios are used. An example from the scenarios is shown in Figure B.3: the text *the system plays his (Carlo) preferred song*, is used for detecting the 'PATIENT.preferredSong' parameter of the task *play song*.

B.2.3 Required Services

In order to support the automation of the identified behaviour patterns, the services shown in Figure B.4 were required.

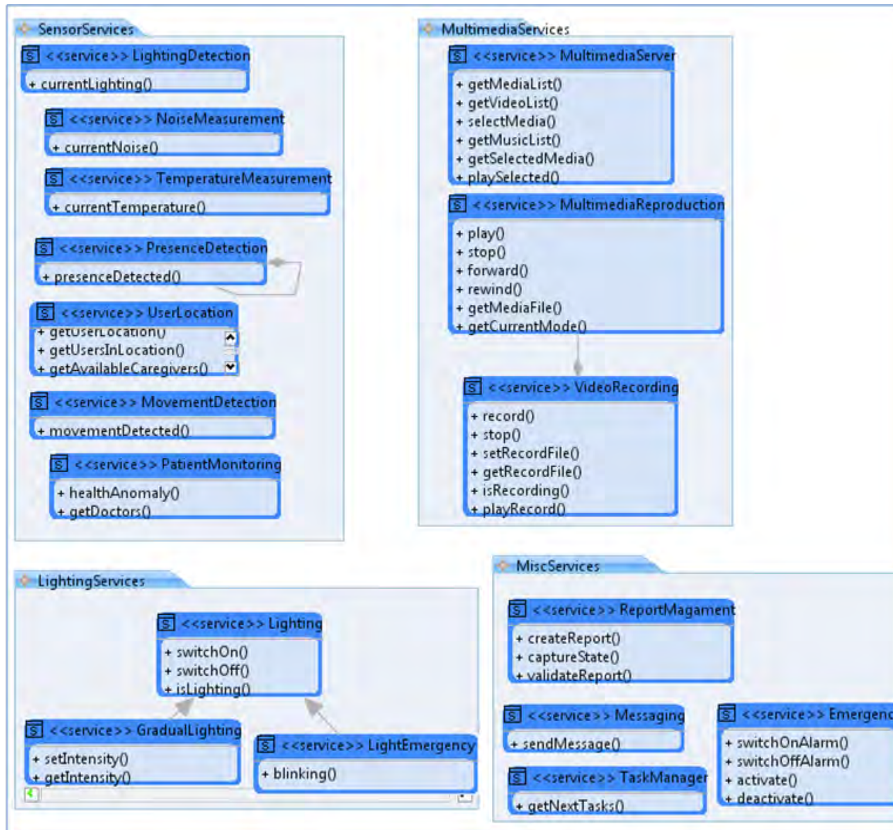

Figure B.4: Services required for the nursing home case studies

www.pros.upv.es

Centro de Investigación en Métodos
de Producción de Software
Universidad Politécnica de Valencia
Camino de Vera s/n
Building 1F
46007 Valencia
Spain
Tel: (+34) 963 877 007 (Ext. 83530)
Fax: (+34) 963 877 359