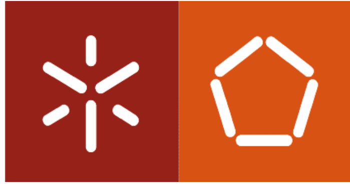

Universidade do Minho
Escola de Engenharia

José Luís Cardoso da Silva

Rapid Prototyping of Ubiquitous Computing Environments

Tese de Doutoramento em Informática

Trabalho efetuado sob a orientação de:

Professor Doutor José Francisco Creissac Freitas de Campos

Professor Doutor Michael Douglas Harrison

Março de 2012

É AUTORIZADA A REPRODUÇÃO INTEGRAL DESTA TESE APENAS PARA EFEITOS DE INVESTIGAÇÃO, MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE;

Universidade do Minho, / /

Assinatura:

Acknowledgments

Many people contributed to this thesis. First of all, I would like to thank my supervisors José Creissac Campos and Michael Harrison for their expert guidance, strong support and continuous encouragement and whose human side I highly appreciated. Sincerely, I feel so lucky to have you as supervisors! This period was an experience which has helped me grow up not only professionally but also as a person and resulted in this thesis that would have not been possible without both of you.

During these last four years, I had the pleasure of collaborating with many people. I would like to thank all Newcastle University members that made my stay in Newcastle more pleasant. I also want to thank friends who I have made in Newcastle, in particular, Sergio Moya, Zhao Ran, Chu Chien, Yousef Abushnak, Abdulrahman, Fahren Bukhari, Paulo and members of *The Globe* which I had the pleasure to meet and which contributed for a agreeable period of my life. Finally, a cherished thank you to Mary Holland for all help and patience.

Many thanks are due to the members of *the Creativity Machine Environment Lab*, *Aware Home Research Initiative Group* and *Ubiquitous Computing Research Group* of the Georgia Technology Institute (USA) for their feedback and suggestions. I would like to specially thank Mario Romero for having provided the *Google Sketchup* model of the *Aware Home* and Gregory Abowd for having hosted me.

Parts of my thesis were published in conferences. I would like to thank my co-authors, all the anonymous reviewers and all people who provided important remarks, comments and suggestions. Also thank you for all members and participants of the APEX project for their contributions.

I would like to thank the Fundação para a Ciência e Tecnologia for their financial support.

All my colleagues and friends, especially João Carlos, Helder and Rafael and, my cousins José Manuel and César who are so important in my life, thank you!

My last words go to my family who always support me, in particular, Susana for her encouragement and for being always available to help me. My parents sacrificed a lot to make sure that I had a better life. To them a few words in Portuguese: *Meus queridos pais, do fundo do coração, muito obrigado por tudo...*

My brother is my model! A very special thank you to João Carlos for being my biggest friend...

Rodrigo is the happiness of my family! Children are a sun in this world...

Finally, a very special thank you to Elisabete for all her continuous encouragement, affectionate support and love which were so precious to accomplish this thesis...

This work is funded by the ERDF through the Programme COMPETE and by the Portuguese Government through FCT - Foundation for Science and Technology, project ref. FCOMP-01-0124-FEDER-015095 and by FCT, under the grant with reference SFRH/BD/41179/2007.

FCT Fundação para a Ciência e a Tecnologia
MINISTÉRIO DA EDUCAÇÃO E CIÊNCIA

Rapid Prototyping of Ubiquitous Computing Environments

Abstract

Ubiquitous computing raises new usability challenges that cut across design and development. We are particularly interested in environments enhanced with sensors, public displays and personal devices. How can prototypes be used to explore the users' mobility and interaction, both explicitly and implicitly, to access services within these environments? Because of the potential cost of development and design failure, these systems must be explored using early assessment techniques and versions of the systems that could disrupt if deployed in the target environment. These techniques are required to evaluate alternative solutions before making the decision to deploy the system on location. This is crucial for a successful development, that anticipates potential user problems, and reduces the cost of redesign.

This thesis reports on the development of a framework for the rapid prototyping and analysis of ubiquitous computing environments that facilitates the evaluation of design alternatives. It describes APEX, a framework that brings together an existing 3D Application Server with a modelling tool. APEX-based prototypes enable users to navigate a virtual world simulation of the envisaged ubiquitous environment. By this means users can experience many of the features of the proposed design. Prototypes and their simulations are generated in the framework to help the developer understand how the user might experience the system. These are supported through three different layers: a simulation layer (using a 3D Application Server); a modelling layer (using a modelling tool) and a physical layer (using external devices and real users). APEX allows the developer to move between these layers to evaluate different features. It supports exploration of user experience through observation of how users might behave with the system as well as enabling exhaustive analysis based on models. The models support checking of properties based on patterns. These patterns are based on ones that have been used successfully in interactive system analysis in other contexts. They help the analyst to generate and verify relevant properties. Where these properties fail then scenarios suggested by the failure provide an important aid to redesign.

Keywords:

Ubiquitous and Context-Aware Computing, Analysis, Modelling, Prototyping, 3D Virtual Environments, User Experience

Prototipagem Rápida de Ambientes de Computação Ubíqua

Resumo

A computação ubíqua levanta novos desafios de usabilidade transversais ao seu desenvolvimento e *design*. Estamos particularmente interessados em ambientes enriquecidos com sensores, ecrãs públicos e dispositivos pessoais e em saber como podem ser utilizados protótipos na exploração da mobilidade e interação, implícita e explicita, dos utilizadores de forma a acederem a serviços desses ambientes. Devido às potenciais falhas do *design* proposto e aos elevados custos associados ao seu desenvolvimento, as características destes sistemas devem ser exploradas utilizando versões preliminares dos mesmos dado que estes podem vir a falhar quando implementados no destino, tornando a sua utilização inaceitável. Essas técnicas são necessárias por forma a avaliar soluções alternativas antes de decidir implementar o sistema fisicamente. Isto é crucial para um desenvolvimento com sucesso que antecipe potencias problemas do utilizador e reduza os custos de *redesign*.

Esta tese descreve o desenvolvimento de uma ferramenta para a prototipagem rápida e análise de ambientes de computação ubíqua como suporte à avaliação de *designs* alternativos. É apresentado a APEX, uma plataforma que junta um servidor de aplicações 3D com uma ferramenta de modelação. Os protótipos baseados na APEX permitem aos seus utilizadores finais navegarem numa simulação 3D do ambiente ubíquo projetado. Desta forma muitas das características do *design* proposto podem ser experienciadas pelos utilizadores. Os protótipos e respetivas simulações são gerados na plataforma para ajudar os *designers/developers* a entender como é que os utilizadores podem experienciar o sistema. Os protótipos são suportadas através de três camadas: a camada de simulação (utilizando um servidor de aplicações 3D); a camada de modelação (utilizando uma ferramenta de modelação) e uma camada física (utilizando dispositivos externos e utilizadores reais). A plataforma possibilita aos *designers/developers* moverem-se entre estas camadas de forma a avaliar diferentes características do sistema, desde a experiencia do utilizador até ao seu comportamento através de uma analise exaustiva do sistema ubíquo baseada em modelos. Os modelos suportam a verificação de propriedades baseadas em padrões. Estes padrões são baseados em padrões existentes e já

utilizados com sucesso, noutros contextos, na análise de sistemas interativos. Eles auxiliam a geração e verificação de propriedades relevantes. O local onde estas propriedade falham sugere um cenário de falha que fornece uma ajuda importante no *redesign* do sistema.

Palavras-chave:

Computação Ubíqua e Ciente do Contexto, Análise, Modelação, Prototipagem, Ambientes Virtuais 3D, Experiência do Utilizador

Author's declaration

The work described in this thesis resulted in the publication and presentation of papers in national and international peer-reviewed conferences:

1. J. L. Silva, J. C. Campos, and M. D. Harrison, “An infrastructure for experience centered agile prototyping of ambient intelligence,” in *Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems*, 2009, pages 79–84. [1]
2. J. L. Silva, Ó. Ribeiro, J. Fernandes, J. Campos, and M. Harrison, “The APEX framework: prototyping of ubiquitous environments based on Petri nets,” in *Human-Centred Software Engineering. Lecture Notes in Computer Science. Springer*, 2010, vol. 6409, pages 6–21. [2]
3. J. L. Silva, Ó. R. Ribeiro, J. M. Fernandes, J. C. Campos, and M. D. Harrison, “Prototipagem rápida de ambientes ubíquos, ”in *4a. Conferência Nacional em Interacção Humano-Computador (Interacção 2010)*, O. Mealha, J. Madeira, D. Tércio and B.S. Santos editors. 2010, pages 121--128, GPCG. [3]
4. J. L. Silva, J. C. Campos, and M. D. Harrison, "Formal analysis of Ubiquitous Computing environments through the APEX framework," in *Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems*, 2012. (Accepted)

In all cases, I have presented only those aspects of the work which are directly attributed to me.

Contents

1 Introduction **27**

1.1 Motivation 27

1.2 Objectives 30

1.3 Research Questions 30

1.4 Thesis Overview 31

2 Background **33**

2.1 Prototyping Approaches 33

2.2 Modelling Approaches 37

2.2.1 Modelling approaches comparison 38

2.2.2 Coloured Petri nets (CPNs or CP-nets) 47

2.3 Virtual Worlds' Simulation 50

2.3.1 3D application servers 51

2.3.2 3D game engines 54

2.4 Analysis 55

2.5 Conclusions 57

3 The Proposed Approach **59**

3.1 APEX Framework 59

3.1.1 Architecture 59

3.1.2 Multi-layer approach 67

3.1.3 Support for design 68

3.2 Alternative Modelling Approaches 70

3.2.1 User-centred approach 73

3.2.2 Sensor-centred approach 79

3.3 Conclusions 81

4 The Modelling Approach **83**

4.1 Using CPN to generate a simulation 83

4.2 The CPN Base Model 84
4.2.1 The model 84
4.2.2 Modeller's tasks 89
4.2.3 Modelling environment's devices 90
4.3 Modelling and Use of Programmed Avatars 91
4.4 Conclusions 92
5 Prototyping Experience Alternatives 95
5.1 Alternative User's Experiences 96
5.1.1 Second Life viewer 96
5.1.2 Support for importing virtual objects 97
5.1.3 Supporting 3D visualization 99
5.1.4 Supporting multi displays systems 100
5.2 Virtual Environment Creation 102
5.3 Conclusions 103
6 Ubiquitous Environments Analysis 105
6.1 Approach 106
6.1.1 Tool support 107
6.1.2 Patterns 108
6.2 Setting Up the Analysis 109
6.2.1 Model conversion 109
6.2.2 APEXi tool - scenario selection and *small colour sets* initialization 112
6.2.3 Reachability graph and analysis 120
6.3 Property Specification Patterns for Ubicomp Environments 127
6.3.1 The consistency pattern 128
6.3.2 The feedback pattern 129
6.3.3 The reachability pattern 131
6.3.4 The precedence pattern 132
6.3.5 The completeness pattern 133
6.3.6 The reversibility pattern 134
6.3.7 The possibility pattern 136

6.3.8 The universality pattern ... 136

6.3.9 The eventually pattern ... 137

6.4 Alternative Analysis ... 138

6.5 Programmed Avatars ... 139

6.6 Conclusions ... 140

7 Examples ... 143

7.1 Smart Library ... 143

7.1.1 The model ... 144

7.1.2 Instantiating property templates ... 147

7.1.3 Checking the model using the SS tool ... 149

7.1.4 The prototype ... 154

7.2 Aware Home ... 155

7.2.1 The model ... 156

7.2.2 Instantiating property templates ... 159

7.2.3 Checking the model using the SS tool ... 160

7.2.4 Checking non-functional properties ... 164

7.2.5 The prototype ... 166

7.3 Serious Games Development ... 167

7.4 Conclusions ... 169

8 Evaluation of the Prototyping Approach ... 173

8.1 Example ... 173

8.2 Relevant User Study Techniques ... 174

8.3 Process and Questionnaire ... 176

8.4 Results ... 177

8.5 Conclusions ... 178

9 Conclusions ... 179

9.1 Answers to Research Questions ... 179

9.2 Summary of Contributions ... 180

9.3 Discussion ... 182

9.4 Threats and limitations ... 183

9.5 Future Work ... 183

Bibliography ... 185

Appendix A: CPN Base Model ... 193

Appendix B: CPN Analysis Model ... 197

Appendix C: Evaluation Questionnaire ... 199

Appendix D: Exercises Proposed During the Evaluation ... 201

Appendix E: APEX Brief User Guide ... 203

List of Abbreviations

2D Two-Dimensional

3D Three-Dimensional

APEX rApid/Agile Prototyping for user EXperience

API Application Programming Interface

CAVE Cave Automatic Virtual Environment

CPN Coloured Petri Nets

CSP Communicating Sequential Processes

CTL Computational Tree Logic

DLL Dynamic-Link Library

FCT Fundação para a Ciência e a Tecnologia (Foundation for Science and Technology)

ICO Interactive Cooperative Objects

GPS Global Positioning System

GUI Graphics User Interface

HCI Human-Computer Interaction

HyNets Hybrid high-level Nets

LAN Local Area Network

LSL Linden Scripting Language

OSG OpenSceneGraph

OSMP Open Source Metaverse Project

PDA Personal Digital Assistant

RFID Radio-Frequency IDentification

UBICOMP UBIquitous COMPuting

UML Unified Modelling Language

UX User eXperience

VE Virtual Environment

VR Virtual Reality

WIMP Windows, Icons, Menus, Pointer

List of Figures

FIGURE 2.1: INFORMAL PROPOSED APPROACH 37

FIGURE 2.2: ASUR++ MODEL (USER ARRIVING AT AN ENTRY GATE) 39

FIGURE 2.3: HYNETS MODEL (USER ARRIVING AT AN ENTRY GATE) 41

FIGURE 2.4: FLOWNETS MODEL (USER ARRIVING AT AN ENTRY GATE) 42

FIGURE 2.5: CPN GRAPHICAL SYNTAX 48

FIGURE 2.6: CPN SIMULATION 50

FIGURE 3.1: APEX ARCHITECTURE 60

FIGURE 3.2: PHYSICAL ARCHITECTURE OF THE APEX FRAMEWORK 64

FIGURE 3.3: BLUETOOTH APPLICATION CLIENT INSTALLED IN A SMART PHONE 64

FIGURE 3.4: OBJECT' IDENTIFIER ACCESSIBLE IN THE VIEWER PANEL 66

FIGURE 3.5: OBJECT MOVEMENT LSL SCRIPT 69

FIGURE 3.6: THE PROCESS 70

FIGURE 3.7: ON/OFF DEVICE ALTERNATIVE MODELS. 1- ALL SEMANTIC IN THE TOKENS, 2 - ALL SEMANTIC IN THE STRUCTURE 71

FIGURE 3.8: USER-CENTRED SMART LIBRARY MODULE 74

FIGURE 3.9: USER-CENTRED SCREEN MODULE 74

FIGURE 3.10: GENERAL EARLY INITIALIZATION MODULE 75

FIGURE 3.11: MODULE FOR AN ENTRY GATE 76

FIGURE 3.12: *ISARRIVINGTOGATEAREA* FUNCTION 76

FIGURE 3.13: GENERIC MODULE FOR ACQUIRING USERS' DATA 78

FIGURE 3.14: PRIORITIES OVER TRANSITIONS 78

FIGURE 3.15: APEX SENSOR-BASED MODELLING APPROACH. PARTS: A) PRESENCE SENSOR MODEL B) LIGHT SENSOR MODEL C) MODELS DEPENDING OF SENSOR'S VALUES 80

FIGURE 4.1: GENERAL EARLY INITIALIZATION MODULE 84

FIGURE 4.2: GATE MODULE 86

FIGURE 4.3: BOOK'S LIGHT MODULE 87

FIGURE 4.4: *COLOUR SET* DECLARATION (LIBRARY EXAMPLE) 90

FIGURE 4.5: PROGRAMMED AVATAR'S MODULE 93

FIGURE 5.1: SECOND LIFE VIEWER 97

FIGURE 5.2: RENDERING OF *MESHES* WITH MESH PROJECT VIEWER 98
FIGURE 5.3: RENDERING OF *MESHES* WITH A VIEWER WHICH DOES NOT SUPPORT *MESHES* (THE OBJECTS PRESENT ARE NOT MESHES)..98
FIGURE 5.4: DALE'S SL VIEWER IN ANAGLYPH STEREO MODE..100
FIGURE 5.5: ACTIVE STEREO - SHUTTER GLASSES BEHAVIOUR..100
FIGURE 5.6: A CAVE 101
FIGURE 5.7: DATA AND DISPLAY SYNCHRONIZATION WITH CAVESL 101
FIGURE 5.8: CAVESL WITH 3 CLIENTS RUNNING AT THE SAME TIME (ADAPTED FROM [82]) 102
FIGURE 5.9: LINKAGE OF THE ELEMENTS COMPOSING A CHAIR 102
FIGURE 5.10: SMART LIBRARY PROTOTYPE CREATED IN THE OPENSIMULATOR SERVER 103
FIGURE 5.11: MESH UPLOAD THROUGH THE PROJECT MESH VIEWER..104
FIGURE 6.1: REACHABILITY GRAPH 107
FIGURE 6.2: *DATATYPEREADING* NON-DETERMINISTIC MODULE..111
FIGURE 6.3: *DATATYPEREADING* DETERMINISTIC MODULE 111
FIGURE 6.4: APEXI TOOL CONNECTION TO THE APEX *BEHAVIOURAL COMPONENT* 113
FIGURE 6.5: APEXI INTERFACE..115
FIGURE 6.6: APEXI AND CPN MODEL CONNECTION MODULE..119
FIGURE 6.7: MODULE WHICH INITIALIZE SOME *SMALL COLOUR SETS* (E.G. *USERSIDS*, *OBJIDS*) 119
FIGURE 6.8: REACHABILITY GRAPH 121
FIGURE 6.9: APEXI SELECTED VALUES..121
FIGURE 6.10: REACHABILITY GRAPH NODE CONSULTATION..122
FIGURE 6.11: CPN LIGHT MODULE 123
FIGURE 6.12: SPECIFICATION OF ELEMENTS IN THE APEXI INTERFACE: A) ONE USER; B) ONE LIGHT; C) ONE PRESENCE SENSOR; D) USER MOVEMENT SIMULATION; E) ONE TIME SENSOR 124
FIGURE 6.13: LIGHTS' ON QUERY 125
FIGURE 6.14: LIST OF NODES WHERE THE LIGHT IS ON 125
FIGURE 6.15: LIGHT TURNS OFF PROPERTY..125
FIGURE 6.16: INSERTING TWO PRESENCE SENSORS IN THE APEXI INTERFACE 126
FIGURE 6.17: CONSISTENCY/FEEDBACK PROPERTY ALGORITHM SKELETON 130
FIGURE 6.18: REACHABILITY PROPERTY ALGORITHM SKELETON 132

Figure 6.19: Precedence property algorithm skeleton 133
Figure 6.20: Completeness algorithm skeleton 134
Figure 6.21: Algorithm skeleton for the identification of the nodes before and after Q............ 135
Figure 6.22: Function to provide all nodes of the reachability graph 136
Figure 6.23: Universality property algorithm skeleton 137
Figure 6.24: Eventually property algorithm skeleton 138
Figure 7.1: User's PDA book direction module 146
Figure 7.2: APEXi - selected values 150
Figure 7.3: Book's light behaviour property (concrete feedback algorithm instantiation) 151
Figure 7.4: Notification property (concrete reachability algorithm instantiation) 153
Figure 7.5: Book's lights system 154
Figure 7.6: People's reading view 155
Figure 7.7: Aware Home floor plan (without furniture) with inserted sensors (one presence sensor and one environment sensor present in each number) 156
Figure 7.8: Aware Home 3D environment 157
Figure 7.9: Parents' alert system behavioural model 158
Figure 7.10: Air quality alert system 158
Figure 7.11: Parents alerted property (feedback) 162
Figure 7.12: Parents alerted property (reachability) 163
Figure 7.13: Physical property (presence sensor distance) 165
Figure 7.14: Aware Home alert system user experience 166
Figure 7.15: Asthma trigger parent's alert via their PDA 167
Figure 7.16: *TriggerHunter* game snapshot (adapted from [95]) 168
Figure 7.17: *TriggerHunter* game asthma trigger management 169
Figure 7.18: Game module 170
Figure 7.19: Trigger Hunter game using APEX - asthma trigger discovered 171
Figure 7.20: APEX - iterative cycle of prototyping (design, test and analysis) 172
Figure 9.1: APEX fields 181
Figure A.0.1: *CPN base* initialization module 193
Figure A.0.2: *CPN base* data type reading module 194

FIGURE A.0.3: *CPN BASE* UPDATE MOVEMENT SENSORS MODULE 194
FIGURE A.0.4: *CPN BASE* UPDATE PRESENCE SENSORS MODULE 195
FIGURE B.0.1: *CPN ANALYSIS* DYNAMIC OBJECTS UPDATE MODULE 197
FIGURE B.0.2: *CPN ANALYSIS* LIGHT SENSORS UPDATE MODULE 198
FIGURE B.0.3: *CPN ANALYSIS* TIME SENSORS UPDATE MODULE 198
FIGURE C.0.1: APEX QUESTIONNAIRE - FIRST PART 199
FIGURE C.0.2: APEX QUESTIONNAIRE - SECOND PART 200
FIGURE E.0.1: INSTALLATION STEPS DIAGRAM 204
FIGURE E.0.2: SENSOR'S ATTRIBUTES 206
FIGURE E.0.3: DYNAMIC OBJECT SCRIPT ASSOCIATION 207
FIGURE E.0.4: EXECUTION OF COMMANDS IN THE VIEWER 208

List of Tables

TABLE 2.1: PROTOTYPING APPROACHES COMPARISON..36

TABLE 2.2: MODELLING APPROACHES COMPARISON 45

TABLE 2.3: 3D APPLICATION SERVERS 54

Chapter 1
Introduction

Ubiquitous computing (ubicomp) was defined in 1988 by Mark Weiser as “machines that fit the human environment instead of forcing humans to enter theirs” [4]. Ubicomp is an evolution of the desktop paradigm of human-computer interaction (HCI). In ubicomp environments, computing is inserted into our environments. This means that these environments possibly do not require active attention because the information is transmitted automatically. These environments pose new challenges for designers and developers mainly because of the wide range of computer science fields involved: distributed computing, sensor networks, mobile computing, HCI and artificial intelligence. In recent years ubicomp systems have become more widespread. Partly as a consequence of device availability the reality of a ubicomp world is becoming more imminent. Ubicomp technologies include smart phones that incorporate many types of sensors, RFID tags and GPS, as well as interactive whiteboards which are used to populate ubicomp environments. This thesis is concerned with the rapid prototyping of ubicomp environments.

The research questions identified during the initial phase of this thesis, and the proposed solutions, led to the proposal of a project[1] which was funded by the FCT (Fundação para a Ciência e Tecnologia).

1.1 Motivation

This thesis focuses on ubicomp designed to enhance physical environments by using “spaces” augmented with sensors, dynamic objects including public displays, and personal devices. Dynamic objects react to interactions and to context changes as well as providing services to

[1] APEX project: http://ivy.di.uminho.pt/apex (last accessed: 9 February 2012).

users in the environment. Of particular interest in these systems is the way that the user interacts with the environment, as a result of both explicit interaction with the system and implicit interactions that arise through changes of context. Here, context could include location, or the steps that have to be taken by a user to achieve some goal (for example check-in, baggage screening, passport control, boarding card scanning).

The experience of checking into an airport can be improved by providing information to travellers when and where they need it. Frustrating delays can be removed through the appropriate use of personalised information. The experience of using a library can be improved by providing personal and clear information about the location of the shelf in a large library where the required book is located. *Experience* is difficult to specify as a clear and precise requirement that can be demonstrated of a system. It is difficult to measure and to obtain early feedback about whether a design will have the required effect to produce a given experience.

Technologically enhanced environments have the potential to transform sterile built environments into places in which people can be in harmony with the environment and its purpose if appropriately designed. Ubiquitous computing poses new challenges for designers and developers of interactive systems. Early prototyping and simulation of ubicomp environments is likely to result in reducing development cost by allowing the assessment of alternatives before expensive development. Because these systems *immerse* their users, the effect they have on the users' experience is an important element contributing to the success of a design. Experience becomes an important characteristic in addition to more traditional notions of usability.

Testing ubicomp systems in real environments suggests the quality issues in ubicomp environments include more than just usability [5]. Essential also is the concern of UX (user experience) [6]. Being able to evaluate and analyse ubiquitous environments as well as providing user experience early, before deployment, is the topic of concern here. The work is based on the presumption that prototypes can be used to explore the impact of a design on users as they move and interact, accessing services within these environments. Prototypes are also used to analyse ubiquitous environment behaviours. To avoid unnecessary development cost, early designs and solutions are explored in this proposal through model-based prototypes explored within a virtual environment. A prototyping framework (APEX) that uses formal

models and binds them to a 3D simulation is the subject of this thesis. The APEX framework supports prototyping and simulation by providing:

- rigorous models of the system behaviour including sensors, dynamic objects and mobile display devices;
- a 3D simulation of the environment created in a virtual world;
- the animation of the 3D simulation based on behavioural models that are also developed within the framework;
- a means of connecting external (physical) devices to the virtual world via Bluetooth. Users can interact either by manipulating physical handheld devices or by controlling avatars located in the virtual world;
- analysis support through the models.

Currently there are no techniques that can be used to analyse specifications against different notions of experience (for a discussion, see [7]). The facets of APEX provide three layers of behavioural representation moving from the abstract to the concrete:

- a modelling layer expressed in terms of formal models;
- a simulation layer programmed or encoded in the 3D simulation;
- a physical layer where external devices are connected to the simulation.

Each layer supports a specific type of evaluation:

- analysis of the model i.e., systematic exploration of the environments behaviour, through analysis of the models (in the modelling layer);
- observation of virtual objects' behaviour, and user reaction to them, within a virtual world (in the simulation layer);
- observation of real objects (for example, actual smart phones) connected to the virtual world, and users' reaction to them (in the physical layer).

Each layer captures a different view on users. Evaluation can be carried out by observing actual behaviour of users within the environment, interacting with the simulation at the physical layer. Alternatively, users can be represented and simulated by avatars that are user representations in the 3D simulation. Finally, user behaviour can also be captured abstractly as tokens within the model. A goal of using a formal model is that a formal analysis of the design can be used to complement the exploration of the design via actual users. While the latter provides valuable feedback about user experience, it is not exhaustive in terms of all the

possible interactions between the numerous components of the ubicomp environment. Hence an exhaustive analysis of the environment's alternative behaviours is desirable.

1.2 Objectives

APEX is designed to satisfy three goals. The first is that it should enable the rapid development of prototypes. A software tool is required that facilitates the development of prototypes, while simultaneously providing the hooks for the target system. The second goal is that a 3D environment can be used to construct simulations that can be explored realistically by users. 3D Application Servers, such as Second Life™[2], Open Wonderland[3] or OpenSimulator[4], provide a fast track to developing virtual worlds. An alternative would be game engines (e.g. OpenScenceGraph[5]). The third goal is an approach to modelling ubicomp environments aiming to create some of the texture of the environment for evaluation purposes. We are interested in creating prototypes of ubiquitous environments from models of envisaged systems. A benefit of this approach is the integration of the modelling approach with analytical approaches, to provide leverage relating to properties of ubiquitous environments that are relevant to their use. The satisfaction of these goals will be demonstrated throughout this thesis.

The development of a framework that assists developers and designers in the efficient development of ubicomp environments is the main objective of this work. The costs involved in the development of these systems for them to be ready for user testing can be very high, consequently there is an opportunity to create a framework which will reduce these costs while providing a way of experiencing the system before physical deployment. Being able to guarantee specific properties of the system and providing analysis and reasoning methods are also features which help developers in the rapid and effective prototyping of these systems.

1.3 Research Questions

The overarching goal therefore is to investigate whether:

[2] Second Life: http://secondlife.com/ (last accessed: 15 November 2011)
[3] Open Wonderland: http://openwonderland.org/ (last accessed: 15 November 2011)
[4] Opensimulator: http://opensimulator.org/ (last accessed: 15 November 2011)
[5] OpenSceneGraph: http://www.openscenegraph.org (last accessed: 15 November 2011)

The ubicomp environment development process can be made easier thereby reducing costs, providing early experience and automated analysis support.

This goal raises a number of consequent issues that needs to be addressed. Firstly an adequate ubicomp environment representation model should be identified which facilitates development. Secondly, these environments should provide experience during the early phases of development. Thirdly, they should be able to be analysed through automated mechanisms enabling the verification of properties. Having this in mind the following primary research questions needs to be answered:

Question 1: can a formal model represent ubicomp environments?

Question 2: can ubicomp environments prototypes address features with the potential to assess user experience without physical deployment?

Question 3: can ubicomp environment be analysed in the early phase of development providing evaluation results at different levels?

They will be addressed throughout the thesis.

1.4 Thesis Overview

As stated the main contribution of this thesis is to provide an approach for the rapid prototyping of ubicomp environments. A framework has been developed, providing early experience of the prototyped environment as well as analysis support. The approach will be presented and described via some examples. Finally the results of an evaluation of the tool with potential developers will be outlined. The dissertation document is structured as follows:

- Chapter 2 - *Background*: examines the current state of the art and identifies current needs regarding rapid ubicomp prototyping with a particular emphasis on prototyping, modelling and simulation approaches;
- Chapter 3 - *The Proposed Approach*: presents the approach used for the rapid prototyping of ubicomp environments. The architecture and features of the developed framework are presented. Additionally, two alternative approaches for modelling are outlined. A more thorough description is provided in Chapter 4;
- Chapter 4 - *The Modelling Approach*: presents the selected modelling approach and guidelines to be followed in the modelling of new ubicomp environments. Additionally, the modelling and use of programmed avatars is described;

- Chapter 5 - *Prototyping Experience Alternatives*: presents alternative solutions that are used to provide a more complete and immersive experience to users. Being an essential component of the experience provided, the virtual environment creation process is also presented;
- Chapter 6 - *Ubiquitous Environments Analysis*: introduces the process to accomplish analysis of ubicomp environments prototypes through APEX. The process is based on the use of property patterns that are described. The use of programmed avatars and other alternative analysis approaches are also presented;
- Chapter 7 - *Examples*: describes analysis and experience results provided by the developed framework through its application to three concrete example;
- Chapter 8 - *Evaluation of the Prototyping Approach*: presents relevant alternative user study techniques and shows the results of the evaluation of APEX by developers using a combination of two of the stated user study techniques;
- Chapter 9 - *Conclusions*: presents a summary of the contributions of the thesis and discusses some of the results obtained. Finally, directions for future work are pointed out.

Chapter 2

Background

This chapter is divided into five sections. The first introduces alternative approaches found in the literature for the prototyping of ubicomp environments. The second section focuses on the description of possible modelling approaches for these environments. The third section describes alternative simulation tools and the forth describes approaches for the analysis of ubicomp environments. The state of the art of prototyping ubicomp environments and related topics (e.g. modelling and simulation) is summarised in the last section.

2.1 Prototyping Approaches

As stated in Chapter 1 the development of ubicomp environments is a complex task that can lead to common problems involving design, cost and the physical space in which they are situated. Fielding such systems for testing purposes is, in many cases, not feasible (consider a hospital or an airport). The acquisition of adequate resources (e.g. physical devices, sensors), the deployment to its target location, and subsequent experimentation, all have expensive associated costs. Design decisions, once committed to, can be difficult to reverse [8]. There is a real problem with identifying and resolving subtle design issues at an early stage in the design process before too many resources have been invested and too much time elapsed. Several approaches found in the literature are concerned with this problem as well as providing and suggesting solutions for it. Any method that allows a system to be explored or analysed at an early stage may indeed make a strong contribution to the field. Some of the proposed approaches concern ubicomp prototyping. Prototypes provide designers with a way of checking proposed solutions with low investment. However there is a tension between the quality of the results provided by prototypes and the quality of the results provided by the physical

implementation of the final systems. Prototypes should provide results reflecting adequately the aspects of the physical implementation of the system in location.

Prototyping ubiquitous systems (see [9] for a good overview) is mostly concerned with the development of prototypes of isolated devices e.g. UbiWise [10], UbiREAL [11], d.tools [12] or Topiary [13]. The systems are typically to be explored outside the context of the fully integrated system in its proposed setting, see Abowd et al.'s paper in [14] for a useful discussion of this contrast. For example d.tools is used to prototype isolated devices through integrated design, test and analysis. The tool provides the capability to connect physical devices (e.g. sensors, actuators) to the developed models enabling the behaviour analysis of the physical devices through their respective model specifications. The framework supports the whole cycle of prototyping of isolated devices. Topiary enables users to explore prototypes of a context aware application in a real world setting. These prototypes often use Wizard of Oz techniques to avoid the need for physical sensors and actual physical. These systems miss the crucial interplay between device and environment to aid understanding of the prototyped environment [15]. Displays, devices and sensors form an integrated whole that, together with physical characteristics of the environment, contributes to the texture of the resulting system. In the context of healthcare applications, within which one of our examples fits, Kang et al. [16] propose a systematic design tool for context aware systems in a smart home. The context aware framework developed using their tool works as a middleware between sensors and service entities. Their focus is not the prototyping of ubiquitous environments but rather the prototyping of middleware to solve the interoperability problem among sensor makers and healthcare service providers.

3DSim [17], UbiWorld [18], the work of O'Neill et al. [19, 20], and VARU [21] develop simulations of actual environments. While 3DSim and UbiWorld use programming languages to build prototypes, the benefit of the work of O'Neill et al. is that modelling can be combined with simulations. They combine models with a 3D simulation to prototype ubiquitous environments. Vanacken et al. [22] also adopt a model based approach. However their focus is on the detail of the interaction within the 3D virtual environment and not in the development of ubicomp environments. In VARU a prototype of a tangible space, combining virtual and augmented reality, can be used to explore ubiquitous computing. A rendering game engine based on OpenSceneGraph[6] is used to achieve this.

[6] OpenSceneGraph: http://www.openscenegraph.org (last accessed: 15 November 2011)

Activity Studio [23] is a tool for prototyping and in-situ testing of ubicomp application prototypes. It provides support for testing low-cost ubicomp prototypes in experimentally relevant environments over extended periods. Several users can explore the prototype over time and information, either from real sensors or reported by users can be provided to the prototype. The analysis is produced as a result of monitoring user activities. The approach lowers the cost of in-situ testing and deploying of ubicomp prototypes. The work of Sohn et al. (ICap [24]) is also relevant here, it assists developers in the informal prototyping and testing of context-aware applications by providing a tool that allow users to quickly define input and output devices and involves rule based conditions prior to the development of an executable system. Momento [25] is a tool for early-stage prototyping and situated experimentation and evaluation of ubicomp applications but does not provide exhaustive analysis support. It is focussed on ubicomp applications experimentation by experience sampling or other qualitative data rather than using a virtual environment.

A further approach, the work of Pushpendra Singh et al., involves the rapid prototyping and evaluation of intelligent environments using immersive video [26]. The approach provides some advantages by removing the need to develop (virtual) environments. However important features that will provide an adequate infrastructure for prototyping are currently listed as future work. The major omission is that it does not provide users with capabilities to interact with the environment. Unfortunately, publications that demonstrate achievement of these plans cannot be found in the literature.

Different types of prototyping are possible, for example:

1. a single device isolated from its context of use;
2. an application/device and it context of use;
3. the environment as enriched by devices.

While several approaches, aiming at ubiquitous computing prototyping, were identified above, they are mostly focused on helping ubiquitous system designers to identify unwanted behaviour in their system, and to support informed decision making in an iterative design cycle. Some of these approaches focus on ubicomp prototyping applications or isolated devices and not on the prototyping of ubicomp environments as a whole. Some provide a notion of experience but not of the whole ubicomp environment. The work of O'Neill combines 3D simulation with models but their focus is to identify occurrences of unwanted system behaviours. They do not provide exhaustive analysis support. Other approaches, for example VARU, also provide user experience but do not support analysis. Table 2.1 summarizes the

key features of the main solutions presented. Some of the stated solutions are not listed in the table because they share common features of presented solutions. For example the UbiWise solution is not listed because it has common main features with *d.tools*.

	UbiREAL	d.tools	Topiary	3DSim	UbiWorld	VARU	Activity Studio	Momento
Applications/isolated devices prototyping	yes	yes	yes	yes	yes	yes	yes	yes
Unwanted behaviour identification	yes	yes	yes	yes	yes	yes	yes	yes
Ubicomp environments prototyping	no	no	no	yes	yes	yes	no	no
Provide user experience	no	yes	yes	yes	yes	yes	yes	no
Formal exhaustive analysis support	no	no	no	no	no	no	no	no
Whole cycle of prototyping support	no	yes	yes	no	no	no	no	no

Table 2.1: Prototyping approaches comparison

There is an absence of an approach that focuses on the experience that users will have of the design of the whole ubicomp environment, and which supports a formal and exhaustive analysis. An approach satisfying these requirements will provide a distinct advance over the state of the art. To convincingly demonstrate its utility, when compared with the existing frameworks, the desired approach should provide:

- support for the design of the ubicomp environment and to explore alternatives, with a particular emphasis on how users will experience it (including first view experience);
- support for analysis either by simulation (similar to program execution) or by more formal analysis;
- a multilayered development approach similar to d.tools and VARU approaches;
- support for the whole cycle of prototyping (design, experience, test and analysis) similar to the d.tools approach;
- multi-user support and collaborative features (e.g. speaking, chatting) enabling interaction between users.

A possible approach, satisfying these requirements, should combine a simulation which will provide users with a way of experiencing a proposed design with a formal modelling approach to enable systematic and exhaustive analysis.

Computer simulation has become a fundamental component in areas such as mathematics, physics, biology, economics and engineering. Simulations are used to estimate and analyse the behaviour of complex systems [27]. They are also used in the development processes of many types of products (e.g. architecture, automotive industry). In particular 3D simulations are used to validate a design before physical deployment on location. 3D provides a representation that contains more features of the real world than non-3D simulations. In the specific case of ubicomp it should be rich enough to provide users with the impression of being in the deployed physical environment. On the one hand a 3D simulation seems a good candidate to provide users with experience of ubicomp environments. On the other hand the modelling approach should enable the modelling of ubicomp environments, allowing reasoning and providing analysis support both in the modelling and prototyping phases of the development. The abstract idea of the proposed approach is presented in Figure 2.1.

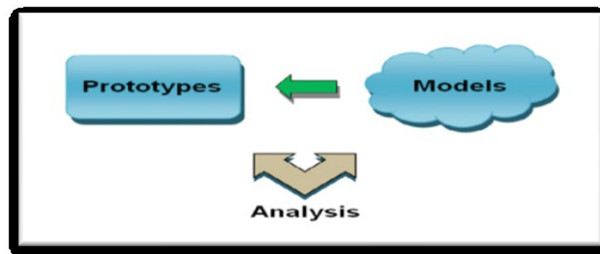

Figure 2.1: Informal proposed approach

In the next two sections several alternatives relating to modelling and 3D simulation are addressed. Comparisons between existing alternatives are presented. Both modelling and simulation approaches which best satisfy the stated requirements are selected.

2.2 Modelling Approaches

Ubicomp environments involve several types of interaction and potentially provide many services to users. Modelling techniques can capture interactions and be used to reason about environment behaviours. We are unaware of a modelling approach specifically developed to deal with ubicomp environments. However the ASUR++ [28, 29] modelling approach was

developed for mobile mixed systems that share some aspects with ubicomp environments. In developing APEX several approaches were considered regarding the modelling of these environments. This section presents a number of candidate modelling approaches found in the literature. These include:

- ASUR++;
- Hybrid high-level Nets (HyNets) [30];
- Flownets [31];
- Interactive Cooperative Objects (ICO) [32, 33];
- Coloured Petri nets (CPN) [34];
- Communicating Sequential Processes (CSP) [35];
- Statecharts [36].

Virtual reality modelling languages such as VRIXML [37] or Web3D languages such as VRML [38] or X3D[7] have not been considered. The goal is not the modelling of virtual reality environments, but to drive ubiquitous environments from models and use the mathematical properties of the model for analysis. For a description of modelling approaches for the virtual reality domain and associated challenges see the work of De Troyer et al. [39].

2.2.1 Modelling approaches comparison

The above modelling approaches are considered and compared in this section. However the goal is not to describe the notations in detail, as that would be tedious, but to compare their main features in relation to the goal of modelling ubicomp environments. The main criteria for assessment are: the presence of a editor, the animation of models and the possibility of hierarchical description, automatic verification of properties and separation of continuous and discrete parts. The next section provides a more detailed description of the notation that better satisfies the requirements.

ASUR++

The ASUR++ notation is an extension of the existing notation ASUR [40]. ASUR was designed to help in the reasoning of Mixed Systems [41]. Mixed Systems are interactive sys-

[7] X3D toolkit: http://artis.imag.fr/Software/X3D/ (last accessed: 30 January 2012)

tems that combine the characteristics of physical and digital worlds. ASUR++ was developed to design mobile mixed systems including features such as spatial relationships between elements [29].

This notation works at a high level of abstraction rather than focussing on a functional behaviour description of system components. ASUR++ characterizes the components and relations of the system. It includes features described as *interaction groups* that enable designers to express spatial relationships between users and entities [29]. An *interaction group* represents a set of entities and channels of the system having common properties that are important for a particular design issue. Many *interaction groups* can be identified for a particular interaction design. Some are applicable to any design and others can be applied depending on the tasks and context. Entities and channels can be grouped based on coherence among properties to generate a coherent effect such as perceptual continuity (e.g. sound and visual) [40]. ASUR++ possesses an editor that enables an architectural view. However the editor does not enable the animation of the models and does not provide support for a hierarchical description and exhaustive analysis. Figure 2.2 shows an ASUR++ model of a user arriving at an entry gate. The gate opens if the user has permission to enter and the screen shows relevant information. The user triggers (==>) an *Input Adapter* (RFID sensor) that exchanges information (→) with the system. Consequently the system exchanges information with two elements, a gate and a screen that are physically associated (==). Finally the gate opens or remains closed and the screen provides relevant information to the user (→).

ASUR++ is appropriate to identify design issues, studying the entities and relations involved and, to think about the transfer of information.

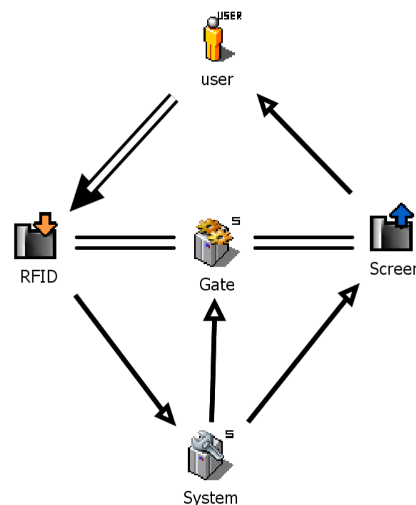

Figure 2.2: ASUR++ model (user arriving at an entry gate)

HyNets

The Petri net notation (also known as place/transition net or P/T net) is a mathematical modelling language and is the base for most of the modelling approaches being considered (e.g. HyNets, Flownets, ICO), see Table 2.2. Petri nets formalism is used to describe distributed systems and to model concurrent computation applied to areas such as software design, workflow management, process modelling, simulation, etc. Several tools are provided to support the development of modelling approaches based on the Petri net formalism (e.g. GreatSPN[8]). Petri nets are strongly focused on the analysis of functional properties.

Hybrid high-level Nets (HyNets) provide a methodology for modelling and simulation of hybrid systems. Hynets combine three modelling approaches:

- a graphical notation to define discrete and continuous parallel behaviour;
- object orientation;
- differential and algebraic equations.

The continuous part of the methodology is used to represent behaviours involving variables whose values vary continuously (e.g. usually associated with physical measurements), otherwise the discrete part is used (e.g. number of users). The object oriented concept enables more expressiveness and improves the management of the information in relation to non object oriented modelling approaches. In order to accommodate the description of processes in which behaviour evolves in time in a continuous way the modelling approach uses differential equations. In HyNets models, continuous behaviour means changing the value of objects according to the equations of transitions. Transitions represent the actions that the system can make. Differential equations change object values and algebraic equations assign values to objects [30]. Figure 2.3 illustrates the modelling of the example of the previous section (user arriving close to an entry gate). The *RFID_detector* class has an infinite capacity (OMEGA) that means that an infinite number of RFID sensors can be present in the model. This class contains the information provided by RFID detectors. The Screen class has the method *proj* that updates the information on the screen. The idea of continuous transitions is to change the values of objects present in places continuously. In this example, the information of the screen is updated continuously (*scr' = proj(R.getInfo())*) with information provided by the RFID detector (*R.getInfo*). The *getInfo* method gives relevant information to be displayed on

[8] GreatSPN: http://www.di.unito.it/~greatspn/index.html (last accessed: 23 February 2012)

the screen. Only when the RFID detector detects the proximity of an RFID sensor (R.detect = 1) the transition *t1* is fired. A detailed description of HyNets can be found in [30].

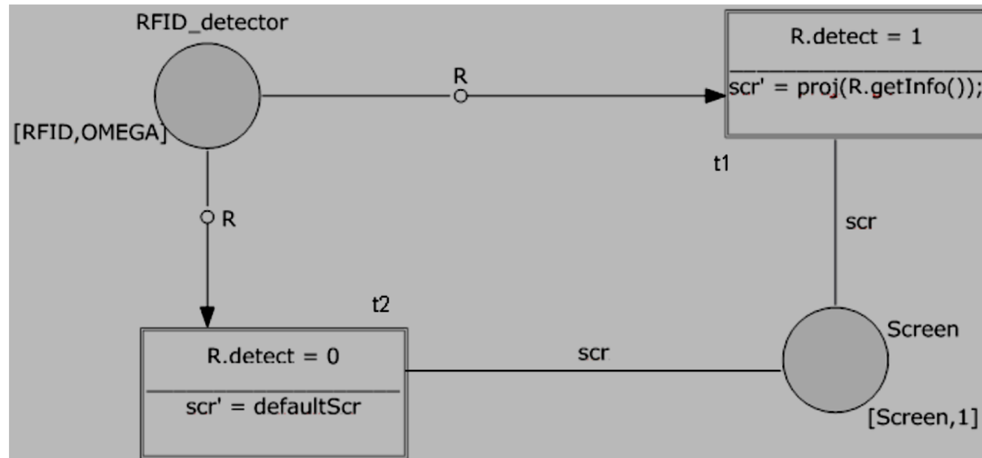

Figure 2.3: HyNets model (user arriving at an entry gate)

Techniques developed for modelling hybrid systems can be used to represent virtual environment interaction, as demonstrated in [42]. Hynets provides hierarchical description support and the separation between continuous and discrete parts but do not possess an editor and do not provide support for exhaustive analysis.

Flownets

Flownets[9] are based on the Petri nets formalism and capture and combine the continuous and discrete parts of user and interactive system interactions in virtual environments. In order to give a general idea of the Flownets formalism, Figure 2.4 illustrates the use of this specification to model the previous example. A clear separation of continuous and discrete parts of the interactive system interaction is made. For instance, the component *RFID detector* receives data continuously (*continuous flow:* =>>). Depending on whether an RFID sensor is or is not detected different behaviour is enabled. This separation is made in a discrete manner (states: *Detect = 1* and *Detect = 0*). With Flownets the states of the interaction and the events that cause the transition states are highlighted, as can be seen in the figure. When the user is detected, the sensor (*detect RFID*) enables the transition from state *Detect = 0* to state *Detect =*

[9] Do not confuse with Flow Nets for hybrid process modelling and control [104]

1. When the system is in state *Detect = 1* the screen is updated (continuous flow). Next the system comes back to state *Detect = 0*, ready to detect another user.

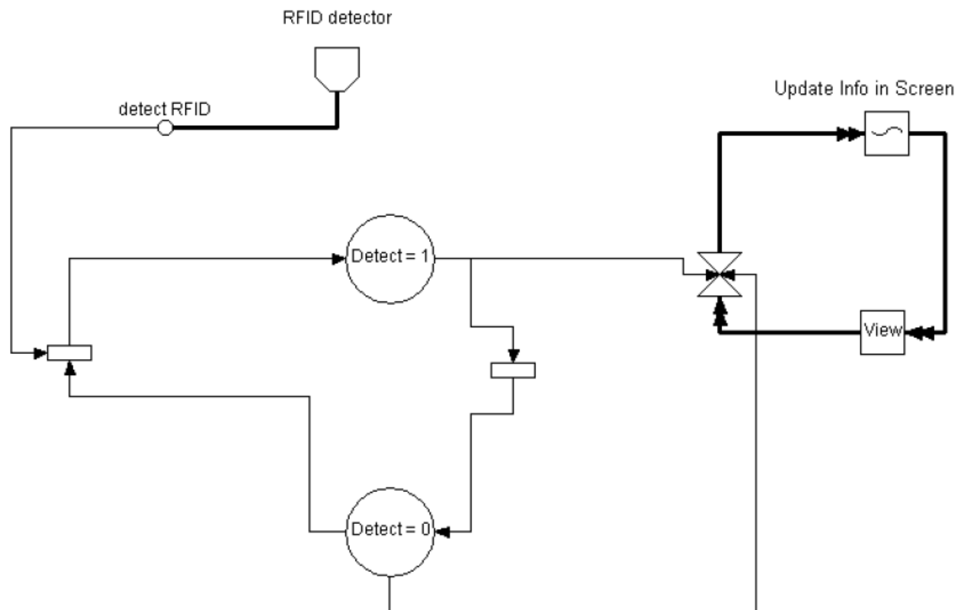

Figure 2.4: Flownets model (user arriving at an entry gate)

Willans [43] proposed an event-based notation to deal with non-WIMP interaction techniques using the Marigold tool. Marigold provides toolset support for the translation from a Flownets design of virtual environment behaviour to an implementation prototype. Using Marigold it is possible to prototype and analyze designs of virtual environments before they are fully implemented. In order to support these approaches, Marigold gives support to automatically check properties and refine designs to a prototype. With Flownets we can define the behaviour of systems and reason about them.

Flownets are supported by an editor that enables a hierarchical description, to animate models and to verify properties. Flownets are appropriate for the design of virtual environment behaviour.

ICO - Interactive Cooperative Objects

This formalism is used to describe interactive systems. Its aim is to provide a precise way of describing, analyzing and reasoning about interactive systems prior to their implementation. Object oriented concepts are used to describe the static parts of the system and high level

Petri nets to describe the behaviour of the dynamic parts The state of an ICO model is defined by the value and distribution of tokens on the places. This follows the state concept used in Petri nets.

The ICO formalism was extended to address new challenges of different application domains. For example, it was extended to support virtual reality applications and multimodal interactive systems. Navarre et al. [32] describe its use to model virtual environment behaviour and multimodal systems including the fusion of several inputs, complex rendering outputs and 3D scenes. These systems possess several inputs and outputs leading to a wide variety of interactions.

ICO is supported by an editor that enables the animation of models and the verification of properties. The formalism does not support the separation between continuous and discrete parts. ICO is appropriate to model interaction techniques and adequate to reason in behaviour and structural levels.

CPN - Coloured Petri nets

CPN is a language used to model concurrent systems enabling the verification of properties on them. It is associated with a functional programming language used to define the data types of the tokens that compose the models and to specify functions and conditions. CPN models enable the verification of properties and simulation (similar to program execution). State space analysis can be used to check standard properties, such as reachability, boundedness, liveness and fairness, as well as specific properties defined using the associated language. The simulation makes it possible to see the behaviour of a model rapidly without detailed human interaction. CPN modelling and analysis is supported by CPN Tools [44, 45]. These tools enable a hierarchical description of the models but without separation between continuous and discrete parts. CPN is explained in more detail in the next section. For a brief description of some of the main CPN qualities see the Coloured Petri net website[10].

CPN is appropriate to model systems that consist of several processes that communicate and synchronize.

[10] CPN qualities: http://www.daimi.au.dk/CPnets/intro/why_cpn.html (last accessed: 30 January 2012).

CSP - Communicating Sequential Processes

Formalisms such as process algebra have been introduced for the same purpose as Petri nets. CSP is a modelling technique, a process algebra using discrete event systems to understand and analyse the dynamics of concurrent systems. By verifying general properties of this formal model, several conceptual difficulties can be revealed at the early stages of the design process, improving the development process. CSP is used to describe patterns of interaction in concurrent systems [35].

Schooten [46] described a modelling technique based on CSP used to model interaction in virtual environments and showed how a prototypes can be generated from the system specification.

CSP is supported by an editor that enables the animation of the models and to verify properties on them. This modelling technique enables a hierarchical description but does not enable the separation between continuous and discrete parts.

CSP is appropriate for reasoning about systems that exhibit parallelism or distribution and consist of multiple components that communicate to exchange information or synchronize (*concurrent systems*).

Statecharts

Statecharts were introduced by Harel in 1987. They are used to model reactive systems and represent a state machine using graphs: nodes denote states and connectors denote transitions. There are currently three main variants of the formalism:

- UML Statecharts;
- Rhapsody Statecharts;
- Classical Statecharts.

There are some differences between these formalisms. In terms of semantics Rhapsody is closer to UML Statecharts than to the Classical Statecharts. UML and Rhapsody do not support simultaneous events or actions [47]. See the work of Crane and Dingel for a detailed comparison [48].

Both Statecharts and Petri nets are generalisations of finite state machines and uses transitions that can enter and leave states. A transition is enabled when all sources are active simultaneously. When the transition is executed all targets became active simultaneously [49].

State is distributed since it depends on the value and state of the nodes which compose the model.

Statecharts are supported by editors that enable the animation of the models. Models can be described hierarchically but there is no separation between the discrete and continuous parts. The automatic verification of properties is also possible.

Statecharts have been widely used to model reactive systems, even in simulation environments where a variant of Rhapsody Statecharts has been proposed [50]. The tools based on Statecharts (e.g. Statemate [51], Stateflow [52]) are more oriented to the software design process and offer for example the capacity to generate code from models [49].

Comparison

Table 2.2 presents a comparison between the different modelling approaches. Note that the classification presented in the table refers to extended versions of Flownets, ICO and Hynets to deal with virtual environments.

	ASUR ++	HyNets	Flownets	ICO	CPN	CSP	Statecharts
Formalism	-	Petri Nets	Petri Nets	Petri Nets	Petri Nets	Process Algebra	Statecharts
Editor	Yes	No	Yes	Yes	Yes	Yes	Yes
Dynamic models	No	No	Yes	Yes	Yes	Yes	Possible [53]
Hierarchical description	No	Yes	Yes	Yes	Yes	Yes	Yes
Automatic verification of properties	No	No	Yes	Yes	Yes	Yes	Possible [54]
Separation of continuous and discrete parts	No	Yes	Yes	No	No	No	No

Table 2.2: Modelling approaches comparison

ASUR++ does not provide a means to model the behaviour of the objects present in the system. This is a weakness for our purpose because we need to be able to model the behaviour of components to drive the interaction. ASUR++ is more appropriate for reasoning at an architectural level.

HyNets is a low level modelling approach. With low-level models we need to specify a model with several details that are close to a low level program. We want to use models at a reasonable level of abstraction that enables us to think about and specify virtual environments without having to worry about specific implementation details. Additionally, HyNets has some disadvantages compared with the other approaches. The main ones are the absence of an editor and a tool that allows for the automatic verification of properties.

Flownets modelling has been successfully used to model the behaviour of a variety of interaction techniques and world objects within virtual environments. However the tool support that is available is less complete when compared with the other approaches (e.g. CPN). Flownets makes a separation between continuous and discrete parts. As we are interested in using models for ubicomp environments, and these are fundamentally hybrid systems, it is important to consider the modelling of both parts.

The ICO modelling approach is dedicated to the specification of interactive systems and is more appropriate to modelling the relation of physical objects present in the system. ICO describes structural and behavioural aspects and the possible interactions that users can have with the application. CPN is mainly appropriate for concurrent systems. This is the case of ubiquitous environments. Both ICO and CPN models can be executed and properties can be verified through the tool support provided.

The thesis of Basten [55] focuses on describing and comparing the Petri net and Process Algebra formalisms. He proposed a method supporting compositional design, combining Petri nets and process algebra. In several aspects these formalisms are complementary. Both formalisms have a mathematical definition and are designed to reason about concurrent systems. Apart from these two common features the formalisms are totally different [55]. Petri nets have a graphical representation in order to make them easier to use and understand for non-experts. Process algebras are a textual formalism. This difference is quite important for our purpose. One of the research questions of the thesis is (Section 1.3) "*can a formal model represent ubicomp environments*". We want modellers to be able to model ubiquitous environments as quickly and easily as possible. In this context the Petri net based formalism is more effective for non-experts than a textual formalism [55]. Since ubicomp environments

are considered to be concurrent systems and both formalisms are designed for these kinds of systems, both formalism can be used for ubicomp environment modelling. See the work of Basten for a detailed comparison [55].

Statecharts and Petri nets formalisms share common concepts having consequently similarities but also differences. The work of Eshuis presents a detailed comparison between the formalisms [49]. This work proposes an algorithm to translate Petri nets into equivalent Statecharts.

In the development of APEX several possibilities were considered regarding the modelling of ubiquitous environments. In the end the choice was made to use CPN because of the substantial set of tools that are available, making it easier to do our own development. Additionally, the simplicity to use and understand by a non-expert provided other benefits. While the language lacks the features of, for example, Flownets or ASUR++ (e.g. continuous and discrete modelling and user's information perception modelling), we believe it provides enough expressive power to suit our purposes and the tools available provides a rich enough modelling, simulation and analysis environment. The continuous behaviour is dealt with via abstractions which also makes it less expensive to analyse.

The ICO modelling approach also provides tools for the simulation and analysis of the model. We believe that this approach can be used successfully as an alternative modelling approach. Statecharts and standard Petri nets are less adequate for the modelling we are aiming at since their scalability is less direct. Adding or removing elements of the environment can be simply reflected in CPN by adding or removing tokens.

2.2.2 Coloured Petri nets (CPNs or CP-nets)

A more detailed description of CPNs will be provided in this section. CPNs extend standard Petri nets by enabling tokens to have a value (in standard Petri nets all tokens are equivalent) which can be manipulated by a programming language CPN ML based on Standard ML (SML) [56]. CPN ML is part of the CPN notation. CPN provides a hierarchy that enables modelling at different levels of abstraction.

CPN Tools is a computer based tool developed to support the graphical representation and use of standard Petri nets, timed Petri nets and CP-nets. Simulation, state space analysis, and invariant analysis are supported by the tool. Simulation is used to animate the models.

State space analysis is used to check standard properties and specific properties defined by analysts.

Models developed using this tool can also use time to evaluate the performance of the systems. CPNs were developed for systems where synchronisation, communication and resource sharing are important features of the modelled system.

"A CPN model consists of a set of modules which each contains a network of places, arcs and transitions. The modules interact with each other through a set of well-defined interfaces, in a similar way as known from many modern programming languages" [57]. Each place can contain tokens that carry data values, called *token colours*. The type of these values can vary in complexity (e.g. a *String*, a *product*, a *record*, etc.) and are specified in the same way as types are specified in programming languages. Each place can only carry tokens of a specified type. This is called the *colour set* of the place. The CPN components (places, arcs and transitions) can have inscriptions associated with them (CPN ML constructs that affect the behaviour of a net). See CPN Tools website[11] for more information. Figure 2.5 represents the graphical representation of the CPN components and elements. See, for example, the *Action* element. The *sendOpenGate* function in it is described in CPN ML within the CPN Tools.

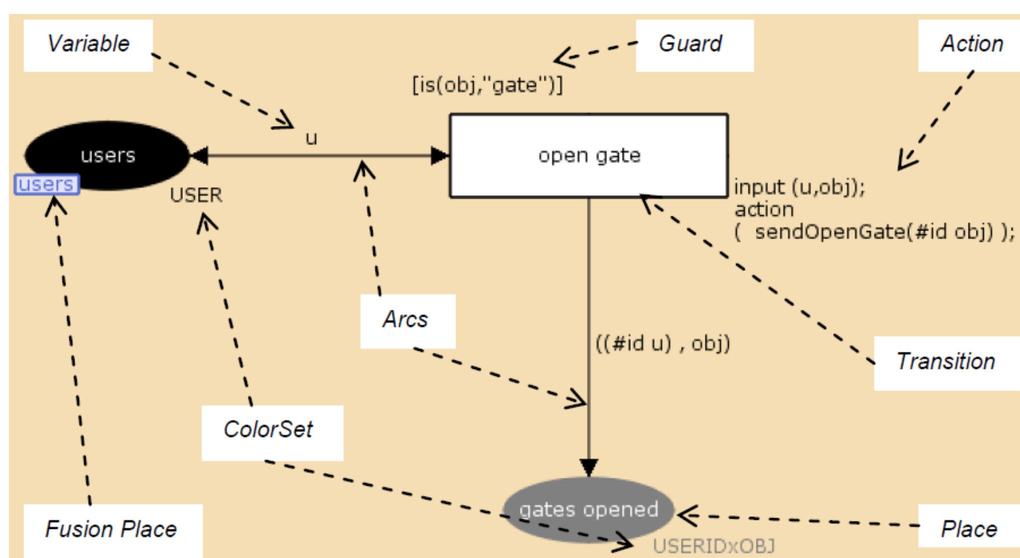

Figure 2.5: CPN graphical syntax

[11] CPN inscriptions: http://cpntools.org/documentation/concepts/colors/inscriptions/start (last accessed: 28 January 2012)

To enable the modelling of hierarchy in these nets two mechanisms are included: *Substitution Transitions* and *Fusion places* [34]. In general, a simplified top level module gives an overview of the system, and other modules connected to it via *Substitution Transitions* and/or *Fusion places* give more detailed information. *Substitution Transitions* make it possible for a more detailed module to be specified at lower levels while providing a clear module abstraction at the top level. These transitions that represent modules are expressed graphically using a box with a double line. This provides a simplified view of a more detailed module and provides the capacity to have multiple layers of detail. The CPN hierarchy also supports places called *Fusion places* that define a set of functionally identical places. The places of a *Fusion place set* can be used in different modules but they are functionally unique. So anything that is happening within a *Fusion place set* also happens to all other places in the set. The graphical representation of these places is illustrated by the place *users* in Figure 2.5.

After the creation of a model it can be executed. Figure 2.6 shows executions of a simple model that simply enables the movement of tokens with values equal to 7 or 8 from place A to place B. The *colour sets* type of the places are Integers (INT). This means that the places can hold integers as tokens. Initially, places can hold tokens where their initial value is determined by evaluating the associated *initialization expression*. The expression *1`7++1`8++1`9* in Figure 2.6 represents an *initialization expression*. In this case, place *A* is initialized with three tokens (integers of the *colour set* INT) with values 7, 8 and 9. The number before the symbol '`' represents the number of tokens with the value which follows (only one token of each value in this case, 1 token with value 7, 1 token with value 8 and 1 token with value 9). The symbol '++' is the syntax used to separate the different tokens, representing each element in the set. In Figure 2.6, the boxes near to the places represent the token's values that are currently held in each place, the number inside the circle indicates the number of tokens held in the place (the number 3 in situation *a,* indicates the presence of 3 tokens in place A). Transitions can have a Boolean expression called a *guard* that enables the execution of the transition when the *guard* is satisfied. The expression *[i=7 orelse i=8]* in the Figure 2.6 represents a guard on transition T. This means that the transition T is enabled only when the variable *i* is equal to 7 or 8.

The CPN simulation binds the tokens to variables of corresponding types. This is done automatically by CPN Tools in a non-deterministic way, i.e. tokens are selected to satisfy the guards, or manually by the analyst. Figure 2.6 shows different states of a simulation. In situation *a,* the variable *i* can be bound to two different values (7 and 8) both satisfying the guard.

In situation *b* the variable *i* was bound to the value 7 and subsequently the token 7 was moved from place *A* to place *B*. In situation *c* only token 8 (the only token present in place A which satisfies the guard) can be bound to the variable *i* in order to enable the transition.

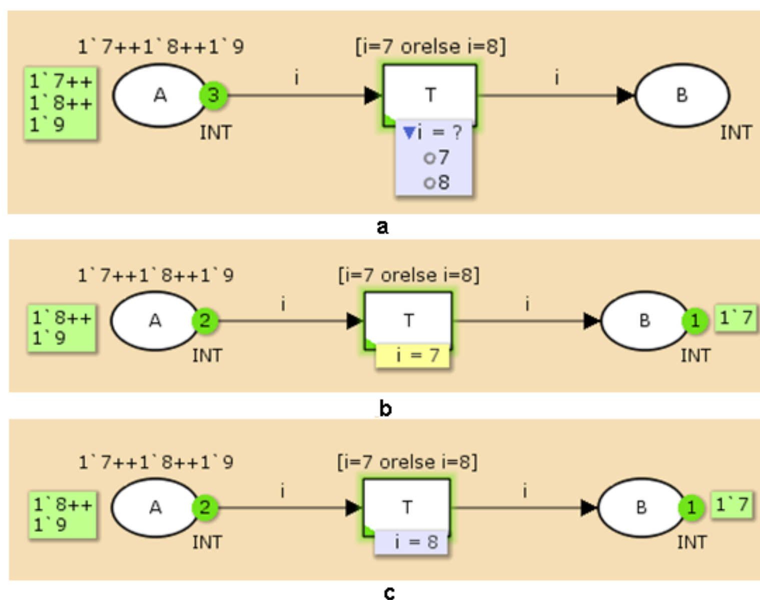

Figure 2.6: CPN simulation

During a simulation many transitions can be enabled at the same time. In these cases only one transition is chosen and executed in each iteration. This selection is automatically done by the CPN Tools. The selection uses a fair algorithm taking into consideration previous selections. However, more recent versions of CPN Tools make it possible to associate priorities to transitions. Consequently the modeller can specify which transition will fire first when there is more than one transition enabled at same time. For a complete description of CPN and CPN Tools see [34, 58, 59].

2.3 Virtual Worlds' Simulation

Simulations enable the exploration of ubicomp environments' usage. Developers and users can navigate and interact within the virtual ubicomp environment getting feedback from it. 3D simulations provide features that immerse users in environments that are intended to provide an experience that is close to the proposed target system. These simulations need to be

sufficiently rich and textured to address usability requirements that depend on the target environment, and to produce an impression of what it will be like to use the final systems once fielded.

This section presents and compares several alternative 3D simulation platforms for virtual worlds. The main criteria for assessment are: the ability to manipulate the virtual world, a sense of occupying space, and the ability to create static/dynamic objects. 3D application servers and 3D game engines will be focused on in particular.

Interactive 3D virtual environments, also called virtual worlds, are computer-based simulated environments. They are widely used in entertainment (e.g. games) but are not limited to them. Many other application domains such as social, medical, commerce, education are examples. Lester and King did experiments which compared face-to-face against 3D virtual worlds results in visual communication classes. The results show that face-to-face students' results are just slightly higher [60]. For instance, many universities, such as open universities, use virtual worlds as an alternative approach to provide education (e.g. virtual classrooms) [61]. In the 3rd quarter of 2010 over 1 billion people worldwide were registered in virtual worlds [62]. Game engines and 3D application servers are existing alternatives to create virtual worlds.

The veracity of evaluations in virtual environments (*ecological* validity) has been addressed in many contexts. For example, Scott addresses it in the medical context [63] and claims that virtual reality has promising ecological validity. Orland et al. [64] considered virtual worlds as representations of landscape realities and as tools for landscape planning suggesting their ecological validity.

2.3.1 3D application servers

3D application servers, which can be accessed through a variety of clients that interact with each other and with the world, are an option to create virtual worlds. Three characteristics are handled by 3D application servers: avatars, regions, and a centralized grid. These worlds are inhabited by avatars which are usually three dimensional representations of humans connected to the environment via the web. A region is a virtual physical place where avatars move and interact. It can be composed of land, water, buildings and/or mountains. These can be created by the avatars within the environment. The grid holds the information about re-

gions which compose the world using processes which can be in different machines. 3D application servers provide a fast track to developing virtual worlds in the sense that the features of these applications enable their rapid creation.

Many 3D application servers can be used to develop virtual environments of which Second Life™[12], IMVU[13], OpenSimulator[14] and Open Wonderland[15] are the most popular. However, there are other possible alternatives, for example Virtual MTV[16], Kaneva[17], Active Worlds[18], Lively[19] and There[20]. For an extensive list of currently available 3D application server see the joakaydia wiki[21] or the work of Freitas [65] (which presents a comparison between alternatives). In some 3D application servers it is possible to own and develop land (e.g. Active Worlds or OpenSimulator). In others it is necessary to pay to own land (e.g. Second Life™). Most virtual worlds provide facilities to chat, walk or play online games. Some of the means of simulating the real world include a market using virtual currency. Second Life™ and IMVU are examples of such systems. Other applications are focused on education and support learning objectives (e.g. Media Grid[22] or project Wonderland[23]). Another difference between existing virtual worlds is the possibility of access to their source code. Open Source Metaverse Project[24] (OSMP), OpenSimulator and project Wonderland are examples of open source virtual world applications. Second Life™ and There on the other hand do not provide access to source code. One additional distinction that separates all these applications is the capability of users to run their own server in a local network. This allows them to maintain their world and provide access to other users. OpenSimulator and OSMP are application servers that offer this feature. Other relevant desirable features include modularity, flexibility and extensibility (e.g. OSMP). These are very important because new functionalities can

[12] Second Life: http://secondlife.com (last accessed: 15 November 2011)
[13] IMVU: www.imvu.com (last accessed: 15 November 2011)
[14] OpenSimulator: http://opensimulator.org (last accessed: 15 November 2011)
[15] Open Wonderland: http://openwonderland.org (last accessed: 15 November 2011)
[16] Virtual MTV: http://virtual.mtv.com/homepage (last accessed: 15 November 2011)
[17] Kaneva: www.kaneva.com (last accessed: 15 November 2011)
[18] Active Worlds: www.activeworlds.com (last accessed: 15 November 2011)
[19] Lively: www.lively.com (last accessed: 15 November 2011)
[20] There: www.there.com (last accessed: 15 November 2011)
[21] 3D application servers: http://wiki.jokaydia.com/page/Vws_list (last accessed: 15 November 2011)
[22] Media Grid: http://mediagrid.org (last accessed: 10 March 2012)
[23] Project Wonderland: //lg3d-wonderland.dev.java.net (last accessed: 10 March 2012)
[24] Open Source Metaverse Project: http://metaverse.sourceforge.net (last accessed: 10 March 2012)

thereby be attached easily by adding new modules. In summary the requirements desired to create a simulation 3D of an ubicomp environment are the following:

- availability;
- building capabilities;
- provide collaborative support;
- provide source code access.
- modularity, flexibility and extensibility;
- possibility to run own server;
- provide dynamic objects.

Table 2.3 sums up the features of the presented application servers.

Not all existing 3D application servers are listed in this table. However the most significant in terms of satisfying the requirements of prototyping ubicomp environments are listed. Other alternatives not mentioned are either similar to the ones presented or less suitable for the purpose.

At the end of the comparative analysis OpenSimulator seems to be the most adequate platform because of its availability, support for creating objects, and the ability to attach behaviours. Its backend, which can be programmed, makes it highly configurable and extensible. These features are the most important to provide support for creating a virtual environment which can simulate ubicomp environments. Several OpenSimulator based projects were and are being developed [66].

As is clear from Table 2.3 Open Wonderland, OSMP or Kaneva also have similar advantages. However they were not selected for a number of reasons. In the case of Open Wonderland it only appeared in 2010 as did the Kaneva 3D app game developer program. The Kaneva developer program allows the possibility of running our own server. The software is modular, enabling developers to create modules to integrate with it. OSMP appeared in 2004 and has all the listed features of OpenSimulator. Unfortunately in 2008 the project was no longer active. The selection of the appropriate platform dates back to 2008 and at that time these three alternatives were not available. Consequently, it was a natural choice to select OpenSimulator in the case of using a 3D application server. Even though they are/were not available these alternatives are listed in the table to provide a complete description of comparable platforms.

	Free to use	Building capabilities	Collaborative support	Open Source	Modular, flexible and extensible	Run own server	Dynamic objects	Period
Second Life™	Yes	Yes	Yes	No	No	No	Yes	2003-now
Active worlds	Yes	Yes	Yes	No	No	No	Yes	1994-now
IMVU	Yes	Paying	Yes	No	No	No	Yes	2004-now
Open Wonderland	Yes	Yes	Yes	Yes	Yes	Yes	Yes	2010-now
OSMP	Yes	Yes	Yes	Yes	Yes	Yes	Yes	2004-2008
OpenSimulator	Yes	Yes	Yes	Yes	Yes	Yes	Yes	2007-now
There	Yes	Yes	Yes	No	No	No	Yes	2003-now* (*broken)
Vivaty	Yes	Yes	Yes	No	No	No	Yes	2008-2010
Kaneva *2010 - 3D app game developer program release	Yes	Yes	Yes	Partially	Yes*	Yes*	Yes*	2006-now
Google Lively	Yes	No	Yes	No	No	No	No	2008-2008

Table 2.3: 3D application servers

2.3.2 3D game engines

A different alternative to creating virtual worlds is the use of 3D game engines. A variety of them are open source, freely available and with different features as a result of their different

purposes (e.g. Blender[25]). Others are proprietary and for commercial use only (e.g. S2 Engine HD[26]) or freeware but without an open source license (e.g. Unity[27]). In this context OpenScenceGraph[28] or OpenSimulator (also considered as a 3D game engine) seem adequate choices, see [67] for a complete list.

Some 3D application servers can be used as game engines. However, the use of 3D application servers has some advantages compared with game engines. Game engines tend to be less oriented to collaborative features such as writing and speaking with other avatars connected to the system. Another advantage of 3D application servers is that they tend to be centralized, while game engines tend to be more distributed. That is a more complex solution for our purpose as it implies several deployments instead of one. They support the creation of virtual environments in real time using world building tools provided by viewers used to connect to them. Using a 3D application server means that a variety of clients, customizable in appearance, can be accessed using multiple protocols. At the same time a virtual world can be maintained in the developer's own server. A disadvantage of 3D application servers is that they provide limited 3D modelling support. They provide basic tool support for object/environment creation but lack advanced support such as provided by game engines. However to compensate for this disadvantage the Mesh Project Viewer[29] has been developed to enable the upload of third party objects to the environment from online libraries (e.g. 3D Google warehouse[30]). Objects created in game engines can be uploaded into 3D application servers using this viewer, avoiding the need of object creation. OpenSimulator was the approach selected to be used in the virtual world exploration and creation process used in the rapid prototyping of ubicomp environments.

2.4 Analysis

A number of techniques within HCI support the analysis of usability of an interactive system from early in its design. These range from paper prototyping and Wizard of Oz techniques, to

[25] Blender: http://www.blender.org (last accessed: 15 November 2011)
[26] S2 Engine HD: http://www.profenix.com/eng/introHD.html (last accessed: 15 November 2011)
[27] Unity3D: http://unity3d.com (last accessed: 15 November 2011)
[28] OpenSceneGraph: http://www.openscenegraph.org (last accessed: 15 November 2011)
[29] Mesh Project Viewer: http://wiki.secondlife.com/wiki/Mesh_Project_Viewer (last accessed: 27 February 2012)
[30] Google 3D Warehouse: http://sketchup.google.com/3dwarehouse/ (last accessed: 27 February 2012)

the development of versions of the systems that can be used during user testing. Other techniques that do not require explicit user testing include the use of expert evaluation techniques such as Heuristic Evaluation and Cognitive Walkthrough.

From a Human-Computer Interaction perspective Nielsen has developed general heuristics for user interface design [68]. Usability evaluation based on heuristics is appropriate for user interfaces (though there are issues, see for example Blandford et al. [69]). However ubicomp environments present challenging usability evaluation problems because they are situated in physical environments and as a result some aspects of the way people interact with them are different from how they interact with more traditional systems [70]. Interaction within the environment may be *explicit* and the devices used for interaction with the system subject to standard usability heuristics for small devices, or it may be *implicit* and arise simply as a result of the user changing their context (for example moving in or out of a room). In both cases each user's context plays an important role. Similar problems happen with other traditional HCI techniques when applied to ubicomp.

A number of evaluation techniques have been developed for dealing with implicit interactions within ubicomp environments. Kim et al. [70], for example, have presented several ubicomp case studies where evaluation has involved making use of physical space. Other evaluation approaches have aimed to provide early evaluation of a partially functional system by using Wizard-of-Oz techniques. Even these more limited approaches involve large resource investments: in one case it involves building physical space for the ubicomp system, and in another developing the system to a partially working level. These costs could be reduced by the application of heuristics to a ubicomp application as explored by Mankoff et al. [71] in the context of ambient displays.

Scholtz et al. [72, 73] have developed a framework for evaluating ubiquitous computing applications. They developed a set of sample measures based on ubiquitous computing evaluation areas to assess whether adequate design principles are satisfied and if the design produces the desired user experience. This framework does not provide an exhaustive means of analysing a developed prototype. Instead the focus is to identify key areas of evaluation and to identify metrics and design guidelines to improve user experience in ubiquitous systems.

Scholtz et al. [74] argue the need to develop interdisciplinary evaluation techniques to address ubicomp properties at early stages in the design. Assessment techniques are required to evaluate alternative solutions before deploying the system. The complexity of a physical

environment where a number of devices are situated, and the added complexity of real world activities, means that it is hard to assess which observations are representative of the use of the system. Likewise it is difficult to assess informally whether characteristics of the system, assessed against specific heuristics, hold across all possible usage scenarios.

The experience of exploring ubicomp environments depends on individual preferences. However some characteristics of user experience can be expressed as properties of the environment. These properties can complement an understanding of experience based on empirical evaluation of the use of a prototype and should be seen as part of a toolset for evaluating a design. We argue that systematic and exhaustive techniques need to be part of an interdisciplinary approach. We follow Mankoff et al. [71] by developing property patterns from existing heuristics. Property patterns have two roles:

i. helping identify interesting properties;
ii. helping verify existing properties.

For example, a property of the *system* requires that there should be feedback for any user of the environment who carries out a particular kind of transaction. This can be expressed as a typical property that takes a standard form. This property pattern would provide the form and would effectively complement evaluation techniques because it provides the option of exhaustive analysis of whether a property is true. This would not be feasible by exploring all possible user behaviours through observation.

2.5 Conclusions

Several approaches have been identified that can be used to prototype ubiquitous systems using virtual environments but limitations were identified (e.g. formal analysis and/or user experience support). Analysing the problem, and existing solutions, the lack of a framework providing together user experience, exhaustive analysis, multi-user and multilayered support, development of the whole cycle of prototyping and collaborative features was identified. Addressing these needs by considering the state of the art revealed several approaches regarding modelling and 3D simulation that were considered to provide an approach satisfying the stated requirements. At the end of the comparison between different modelling and simulation approaches we decided to use a formal modelling approach together with a 3D application server. The modelling and simulation approaches selected were:

- Coloured Petri nets (CPN) - selected mainly because of the tool support provided by the CPN Tools for creation, simulation and analysis. The support to model concurrent behaviour and modularity features were also relevant;
- OpenSimulator (Opensim) - selected mainly because of its extensibility, modularity and the possibility of being able to run an own server. It also possesses a backend that can be programmed.

The idea is to connect CPN with OpenSimulator, providing the benefits of both approaches. It is expected that this approach would satisfy the stated requirements for the rapid prototyping of ubicomp environments. In particular the combination opens the possibility of providing experience to users of the physical target environment and to verify useful properties on it. Support for a multi-user and multilayered prototyping approach covering all phases from design to testing and analysis can also be realised.

Other work, for example Kindler et al. (PNVis), has developed prototype environments combining low level Petri nets with 3D simulation [75]. They simply equip a Petri net with a 3D-visualization making it possible to see the behaviour of a Petri net model through the 3D objects. As stated the proposed approach is much more than the 3D visualization of a Coloured Petri net. APEX aims to provide a first person experience of a ubicomp environment and being able to formally analyse it.

Chapter 3

The Proposed Approach

Early prototyping and simulation of ubiquitous computing environments can reduce development cost by allowing assessment before deployment. This is a particular issue in physical spaces augmented by sensors and dynamic objects including public displays and personal devices. In these cases change as a result of evaluation may require physical reconfiguration of the system. This chapter describes the proposed approach illustrated through two main Sections (3.1 and 3.2). The first section presents the APEX architecture, its multiple layers and the way it supports design. The second section presents two alternative modelling approaches and associated characteristics.

3.1 APEX Framework

APEX enables the flexible development of immersive prototypes based on a 3D Application Server, OpenSimulator, and CPN based behavioural models. The APEX framework also includes a library of virtual sensors (e.g. presence sensors and light sensors) and dynamic objects (e.g. screens, gates, windows, lights) along with their associated CPN models. Elements from the library can be used "off the shelf". This eases the process of prototype development. Details of the framework and decisions made about its structure are presented in this section.

3.1.1 Architecture

Considering the goals behind the development of APEX (see Section 1.2) and the resulting requirements (see Section 2.1), an architecture with four components, each satisfying some

requirements, was created. The whole architecture of the APEX framework was designed aiming at the major goal to provide a framework for the rapid prototyping of ubiquitous environments. The overall architectural view of the framework is presented in Figure 3.1. The four main components are:

1. a *behavioural component*, responsible for managing the behaviour of the prototype, including the description, analysis and validation of the virtual environment's behaviour;
2. a *virtual environment component*, responsible for managing the physical appearance and layout of the prototype, including managing the 3D simulation and the construction of the virtual environment;
3. a *physical component*, responsible for supporting connections to physical external devices, such as smart phones and sensors;
4. a *communication/execution component*, responsible for the data exchange among all components and for the execution of the simulation.

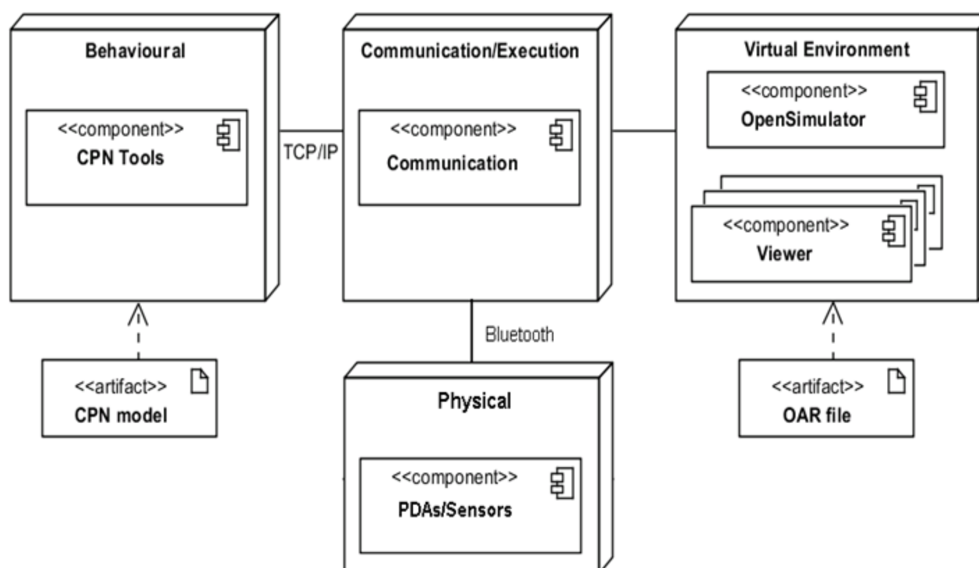

Figure 3.1: APEX architecture

With these components APEX aims to satisfy the stated requirements (e.g. user experience and formal analysis support, see Section 2.1) and supports the whole cycle of prototyping. APEX uses CPN Tools to model the behaviour of the virtual environment. To create a prototype, besides creating the virtual environment, the developer needs to extend a generic

CPN model. This generic model was developed to serve as a base for the modelling of ubicomp environments. Once the CPN model and the environment are created the *communication/execution* component of the framework (see Figure 3.1) binds them together. To achieve this, CPN transitions link the behaviour described by the models to the respective objects in the environment. Additionally, physical components can also be connected to the prototype enabling information to be obtained and/or sent.

After these steps several users can be connected to the simulation using different viewpoints onto the OpenSimulator server. Users can navigate and interact with other connected users as well as the virtual world simulation of the envisaged ubiquitous environment, enabling the evaluation of usability and experience issues with the proposed design.

A description of each component that comprises the APEX framework is now presented.

Behavioural component

This component is responsible for driving the simulation using the information from the model, and for sending/receiving relevant data to/from the virtual environment. It comprises CPN Tools that use CPN models to describe the behaviour of the virtual environment in response to user actions and context changes.

To provide help when modelling, a generic CPN *base model* is provided from which virtual environment models can be derived. The aim in developing this *base model* was to create a generic style of CPN relevant to the modelling of virtual environments. The model consists of modules that:

1. initialise the simulation, and establish the connection between the CPN model, as represented by CPN Tools, and OpenSimulator;
2. receive data (for example sensor's data) from OpenSimulator and use it to update appropriate tokens;
3. describe the behaviour of each device in the system. Sensors and devices are represented separately using different *fusion* places. The sending of information to OpenSimulator is accomplished using provided functions and can be invoked from any module.

This model aims to serve as a basis for the developer to model the behaviour of the desired ubicomp environment. A detailed description of this model and how to extend it to create a prototype of a new ubiquitous environment can be found in the next chapter. Models of

each type of dynamic object/device in the environment (e.g., sensors, displays, personal devices) need to be inserted into the CPN *base model*. Adequate models must either be available or must be created using CPN Tools. Section 4.2.3 will provide a more detailed description of how that can be done. Appendix A presents the whole CPN *base model*.

Virtual environment component

This component sends information about the simulation (e.g. avatar proximity detection) to the behavioural component. The virtual environment component also reflects in the simulation the decisions made by the behavioural component (e.g. open a gate). The virtual environment is composed of the OpenSimulator server and a viewer for each client that is connected to it.

OpenSimulator enables the interactive creation of virtual environments. It provides a texture that enables users to visualise the physical characteristics of the real system. The features of the 3D simulation include location, the viewing aspect and the physics of each of the objects in the environment. Pre-defined environments and objects can be saved/loaded in/from Opensim ARchive files (OAR). All the different entities (object, terrain, textures, etc.) are packaged in these files in the format used by OpenSimulator to keep data within an archive. A number of features are designed to support object/environment creation and to manipulate objects via the viewer. These objects, together with features that support the insertion and manipulation of textures, lighting, animation and sounds, enable the creation of a simulation close to the real proposed system. Pre-defined environments and devices can be used in this creation process. OpenSimulator enables the association of scripts to the world objects using the Linden Scripting Language (LSL), enabling their animation.

The OpenSimulator server is responsible for maintaining the virtual environment information available to viewers. The server enables the connection of several users, each perhaps from a different location, to the same virtual environment via the web through appropriate viewers.

Viewers have two roles. The first role is that they are used as a design tool, to define features of the 3D simulation presented to users. The second role is that they are used as a navigation (end user) tool allowing the user to navigate and interact with the simulated environment. Interaction is achieved both explicitly by a user using (virtual) devices, and implicitly

through changes of context. Adequate viewers include the Mesh Project Viewer[31] or the Linden Lab's Second Life™ viewer[32]. A number of alternative compatible viewers exist (see [76] for a complete list). However, some of these alternative viewers currently only enable the environment exploration without providing any 3D modelling tool. These alternative viewers cannot be used to build new environments but only to explore existing ones.

Physical component

The APEX physical component (see Figure 3.1) allows the connection of external devices such as PDAs and sensors to the framework. A virtual prototype can therefore be combined with the real world, receiving real sensor data and sending information to real physical components. The connection between external devices and the virtual part of the prototype is established using Bluetooth by the *communication/execution* component. Physical devices are used not only to receive information but also to send information to the other layers.

A Bluetooth client application is installed on mobile devices and a Bluetooth server application is installed on the client machines (running in parallel with the viewer – see Figure 3.2). Clients communicate with OpenSimulator via TCP/IP and physical devices communicate with the Bluetooth server. APEX automatically detects mobile devices and links them to relevant avatars in the virtual environment using login information established when users connect the mobile device (see Figure 3.3). For this to work the Bluetooth server must first be selected and the user account corresponding to the desired avatar's device be provided. After a successful connection the mobile device is ready to exchange information with the other components thus improving user immersion. Figure 3.3 shows the Bluetooth Application Client installed on a smart phone running the Windows Mobile Operating System. The use of phones running other operating systems (Windows Phone, iOS, Android, etc.) requires the development of appropriate clients, which poses no particular difficulty.

[31] Mesh Project Viewer: http://wiki.secondlife.com/wiki/Mesh_Project_Viewer (last accessed: 27 February 2012)

[32] Second Life: http://secondlife.com (last accessed: 15 November 2011)

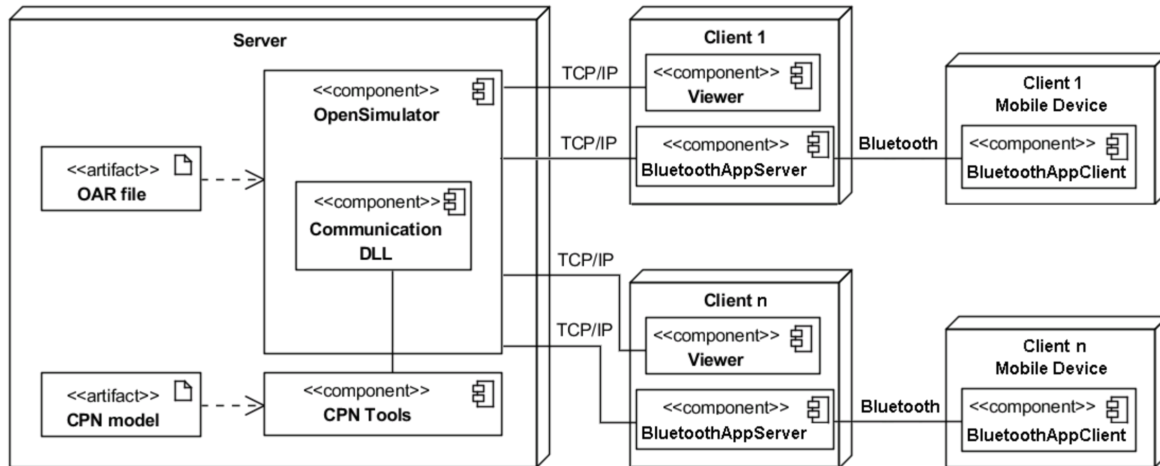

Figure 3.2: Physical architecture of the APEX framework

Figure 3.3: Bluetooth Application Client installed in a smart phone

Users may be located in the physical layer interacting with physical objects, or in the simulation and modelling layer or in a combination of physical/simulation and modelling layers depending on available resources. The interaction with physical devices enables users to experience physical aspects of the proposed target ubiquitous environment (in this case, the actual interface via the smart phone). Results are obtained from user feedback, either immersed in the prototype using a virtual environment on their desktop, or using real elements such as smart phones. In practice mobile devices integrate many sensors (e.g. accelerometer, light, orientation, position, temperature) and provide an easy and uniform solution to acquiring real sensor data.

Communication/execution component

This component is a DLL (dynamic-link library) developed in C# responsible for loading the simulated ubiquitous environment into the OpenSimulator server, and for using the CPN models to drive it. The component is loaded into OpenSimulator at start up and is positioned between the three other components managing the exchange of information between models, the physical devices and the virtual environment. Data exchanged includes strings that give indications to the receiver component about what happened in the sender component. This information enables the update of the receiver component to reflect the changes in the sender component. The information exchange occurs in both directions.

Communication in the CPN Tools is achieved through Comms/CPN [77], a CPN ML library for connecting between CPN Tools and external processes, provided with the CPN Tools. The BRITNeY Suite [78] also enables the communication between CPN models and a Java-based animation package. Comms/CPN is simpler to use than BRITNeY Suite for our case. Unlike Comms/CPN the BRITNeY Suite is more general purpose, providing more features besides the communication package, which make it more complex to use. In order to use Comms/CPN a module must be loaded into the external process. Java and C modules are available with the distribution. However, OpenSimulator modules (DLLs) are developed in C#. No alternatives were found for communication with C# processes, so a new C#/CPN communication module (DLL) was developed. With this development the communication of the CPN models, using the Comms/CPN functions, and C# processes becomes possible. Communication between the model and the *communication/execution component* is achieved based on provided CPN Tools functions of the Comms/CPN library [77].

The developed DLL sends information to CPN Tools when changes in the environment of the physical device happen, and is responsible for changing the environment in response to data sent by CPN Tools. These changes are triggered explicitly through direct user action, or implicitly by a sensor. Actions triggered by the *behavioural component* are thereby reflected in the virtual environment and any physical devices (see Figure 3.6). Additionally, the DLL handles the loading/saving of OpenSimulator objects/environments and the execution of commands invoked by the user in the viewer. When located together with the OpenSimulator server (see Figure 3.2), this DLL is loaded automatically by the server when started. After establishing of communication between CPN model and the simulator, through invocation of

a function in the CPN model (explained in next chapter), the APEX framework is ready to use.

The consistency across uniquely identified multiple representations (e.g. dynamic objects) in the different layers is maintained through the *communication/execution component*. The behaviour described in the *behavioural component* (CPN models) is linked to the *virtual environment* component enabling the animation of the virtual environment. This link between models and corresponding virtual objects is achieved using unique identifiers of the objects present in the virtual environment that are represented by tokens in the CPN models. For instance, to open a gate in the environment, the CPN gates module must indicate in its *open transition code* the identifier (e.g. unique ID) of the gate that must open. Identifiers, object types, positions and other relevant features of all dynamic objects and sensors present in the environment, are automatically loaded into CPN models at the beginning of the APEX execution. The *communication/execution component* synchronizes the values of the dynamic objects and sensors in both components. In the *behaviour component* they are accessible in the tokens values and, in the *virtual environment* component, through the properties panel provided by the viewer associated to each object (see Figure 3.4). For example a dynamic object will contain a unique identifier in the simulation layer that is used to represent it in the modelling layer, and the script linked to the dynamic object will respond to changes in the environment consistent with the state of the CPN model. The illustration of this is presented in this chapter in Section 3.1.3.

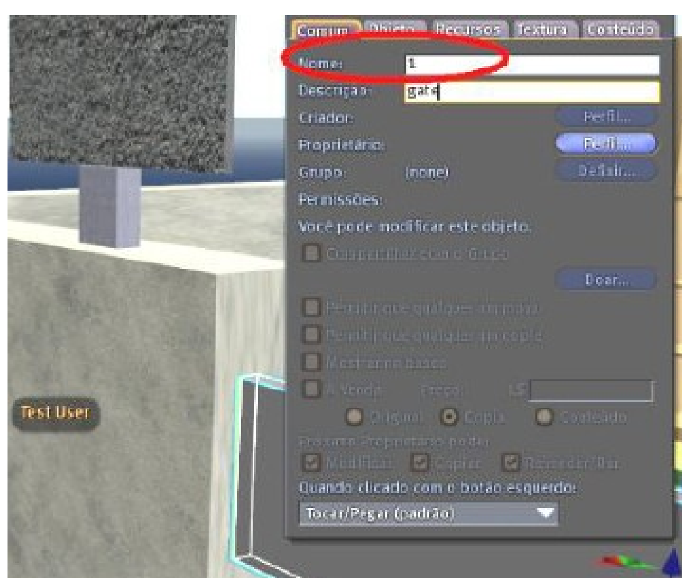

Figure 3.4: Object' identifier accessible in the viewer panel

3.1.2 Multi-layer approach

Prototypes and their simulations are generated using the framework to help the developer understand how the user might experience the system. These prototypes are supported through three different layers: a simulation layer (using the OpenSimulator); a modelling layer (using CPN Tools) and a physical layer (using external devices and real users). APEX allows the developer to move between these layers to evaluate different features, from user experience of using a device, to exhaustive analysis of the ubiquitous environment behaviour.

In an environment that has been developed using APEX users interact with the prototype either directly by manipulating handheld devices as would happen in the deployed system or indirectly by controlling avatars located in the virtual world. Using model-based simulation it is possible both to analyze the system rigorously using the model and to observe user reaction to the system. As previously stated in Chapter 1, each layer supports a specific type of evaluation:

- observation of virtual objects' behaviour, and user reaction to them, within a virtual world (in the simulation layer);
- analysis of the model (in the modelling layer);
- observation of real objects (for example, actual smart phones) connected to the virtual world, and users reaction to them (in the physical layer).

There is also interaction between the layers. The behaviour of users interacting with the simulation at the physical layer can be used to analyze underlying models. Observed behaviour can be represented and simulated by avatars in the simulation layer or captured abstractly as "tokens" within the model. Feedback data from actual use may be collected and "mechanically" analysed, for example by comparing it to expected behaviour [79], using the behavioural model. Programmed avatars may also be developed and used to generate closed simulations in situations where more than one user interacts in the environment. Expressing all the elements and users of the environment at the modelling layer forces the system to be closed, and enables it to be run independently of both the simulation and the physical layers. Hence an exhaustive analysis of the environment's possible behaviours is possible.

The stated multi layer benefits are made concrete, through examples, in Chapter 7.

3.1.3 Support for design

APEX supports both the design and the analysis of ubicomp environments. To achieve this the developer extends the CPN *base model* to respond to changes in the environment consistent with the state of the CPN model. A typical runtime configuration of the framework (see Figure 3.2) involves deploying the OpenSimulator server, CPN Tools, and the *communication/execution* component on a server. Once the CPN model is loaded, the server is ready to allow free exploration and interaction with the virtual environment. Currently this is achieved by means of viewers deployed on client machines.

In addition to exploring the environment, it is also possible to use the viewer to manipulate it, load objects into the environment and to save and clear the environment. This is achieved in the viewer by writing commands in the provided *chat box*. Consult Appendix E to see which commands are available and how to use APEX. The viewer features associated with APEX commands aim to ease the creation and management of virtual environments.

Dynamic objects present in the environment (e.g. gates) are animated using LSL scripts. Figure 3.5 shows an example of an LSL script responsible for moving an object. When executed, this script moves the object it is attached to two units in the positive direction on the X axis. At the second execution the object returns to the original position. The execution of the scripts is triggered by the modelling layer. Scripts are responsible for the visual appearance and behaviour of dynamic objects that arises from moving tokens of the modelling layer from one state to another. The modelling layer is responsible for the logic of the environment described by their state transitions.

Figure 3.6 illustrates the process that leads to the opening of a gate when an avatar comes close to it. Initially the token that represents the gate is on the *Gates Closed* place because no avatar is near the entry gate. Due to space constraints and to improve readability the model presented in this figure is simpler than the real one that is presented in Chapter 7. The model is responsible for specifying the behaviour of the gates. Then following the process (step 1) by user interaction the avatar arrives close to the gate. At this moment the APEX *communication/execution component* detects it (step 2) through the presence sensor located near the gate. This sensor gets the identity of the avatar that is near to it and then this information is sent to the model (step 3). This information leads to a state change of the gate so the token moves to the *Gates Opened* place by the *Open Gate* transition. When this transition is executed the

associated action (*Action Open*) is also executed. In the model this action is composed of a function that sends to the APEX *communication/execution component* the identifier of the gate to open and an indication to execute the script associated with it (step 4). Finally the APEX component searches this object on the environment and when found orders the execution (step 5) of the associated script (see Figure 3.5). The script is executed leading to the gate opening (step 6). The avatar is now able to enter the library.

This process is automatic, developers just have to extend the CPN *base model* with provided modules (when available to model the desired situation) and attach developed scripts to the dynamic objects aiming at the desired behaviour.

```
integer counter = 0;

do_process()
{
   if(counter == 0)
   {
      llSetPos(llGetPos() + <2,0,0>);
      counter++;
   }
   else if(counter == 1)
   {
      llSetPos(llGetPos() - <2,0,0>);
      counter--;
   }
}
default
{
   touch_end(integer i)
   {
      do_process();
   }
}
```

Figure 3.5: Object movement LSL script

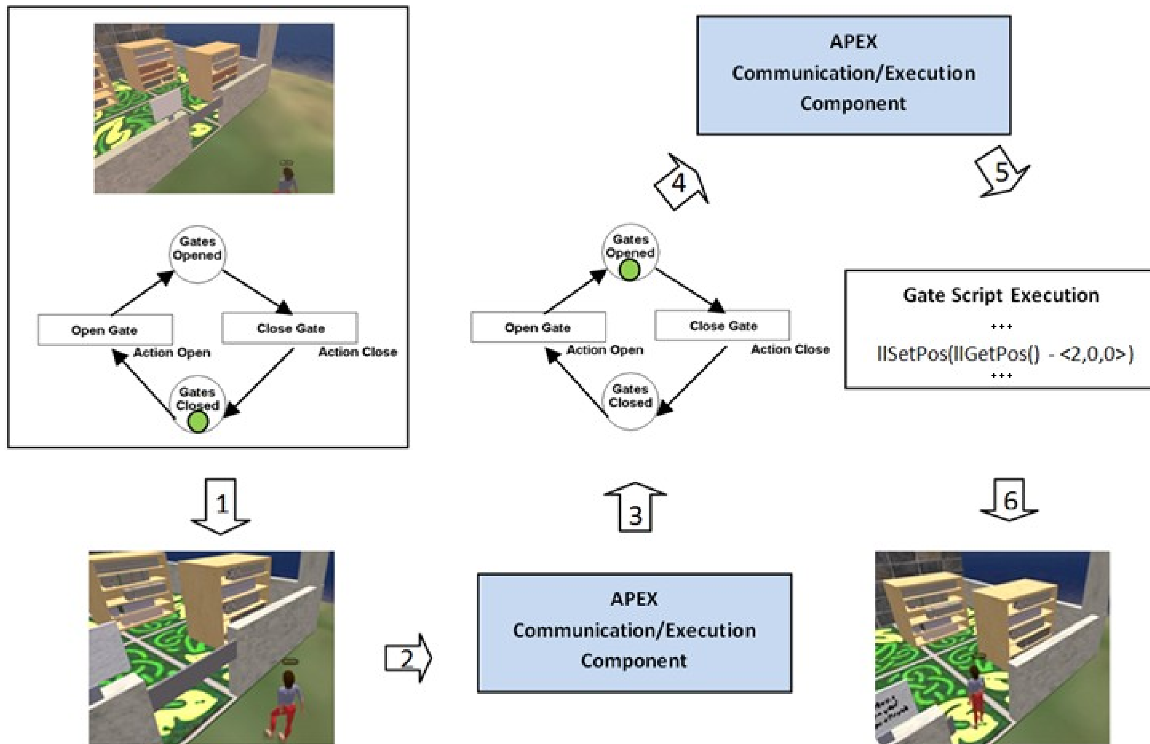

Figure 3.6: The process

3.2 Alternative Modelling Approaches

A generic modelling approach has been developed and is now presented and illustrated through a small example. The example is a smart library environment. This modelling approach enables easier behavioural specification of new ubicomp environment prototypes by providing guidelines to developers. Several alternative approaches were possible. This section shows two of them and the advantages of the selected one.

There are a number of styles of specification that can be achieved using CPN. These styles vary according to the extent to which the semantics of the underlying objects are made explicit in the structure of the CPN specification, or encoded in tokens. Two extremes are possible:

1. All semantics may be placed in the tokens, by this means minimising the number of places in the net;
2. Places may be used to characterize each different relevant situation (user action, context change, etc.), thereby adding transitions that describe aspects of the semantics of the objects explicitly.

A small example clarifies these two extremes. Consider a device that is defined to be in two states (on and off). Two different models capture the characteristics of the device (see Figure 3.7). The first model consists of only one place, and one transition from and to this place. The place holds tokens with a semantics that can represent all the different states of the device. The state of the device will be encoded as an attribute (a *colour*) of the token representing the device. The transition is responsible for changing the *colour* of the token thereby reflecting the new state of the device. In this situation all meaning is in the value of the tokens (see Figure 3.7, tag 1).

The second model is represented by two places each representing a possible state of the device, and by transitions between them (two in this case). No semantics are carried by the token. All meaning is represented by the structure of the model. The state of the device is known by looking to the position of the token, i.e., the place that holds the token (see Figure 3.7, tag 2).

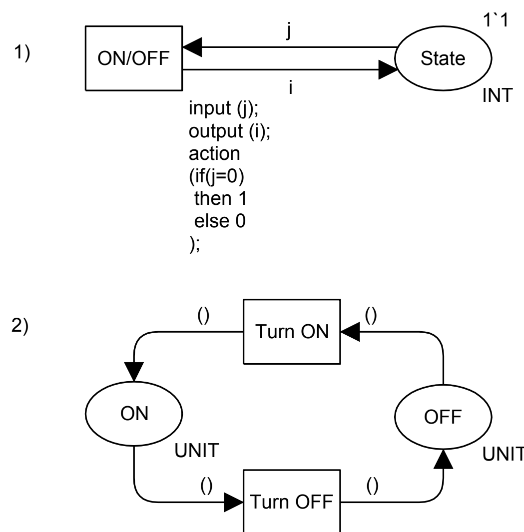

Figure 3.7: On/Off device alternative models. 1- all semantic in the tokens, 2 - all semantic in the structure

In APEX, a mixed approach is used where the states of the dynamic objects (e.g. open, close, off, etc.) are modelled as places and user actions and context changes are modelled as transitions. Each device (sensor or dynamic object) and user is represented in the CPN model as a token in its respective place.

The users and device features (e.g. identifier, position) are modelled as attributes of the tokens. These values are used by CPN ML functions together with instructions (e.g. open,

close) to indicate changes that must be reflected in OpenSimulator. Section 4.2 of next chapter provides a description of how this is done. The guards on the transitions, as well as the functions associated with transitions, are responsible for part of the behaviour of the system. Both of these are modelled in the CPN ML language, so this behaviour is modelled functionally.

This combination gives more expressiveness to the ubiquitous systems modelling while avoiding clutter in the CPN specification. In the process of developing the style of modelling described here two alternatives were considered. A user-oriented approach was first developed where the actual coordinates of the users' position in the virtual world are obtained from the simulation (via the *communication/execution* component when the avatars move), and then forwarded to the CPN models where decisions were made with these values. This approach differs from what happens in physical ubiquitous systems that are sensor-oriented. In these systems the user position is inferred from the values provided by the sensors. This approach therefore is somewhat unrealistic, consequently the validity of some properties of the system is limited. Another limitation of this first alternative is that other sensor information needs to be carried in the user tokens making the approaches less flexible and less consistent. A sensor-oriented approach was also considered.

In summary, the following main criteria were followed to select an adequate modelling approach. The approach has to:

1. be generic;
2. scale;
3. be flexible/modular;
4. be realistic.

The satisfaction of these criteria aims to answer one of the identified research questions (from Section 1.3). *"Can a formal model represent ubicomp environments?"*. A modelling approach, satisfying these criteria, would be a potential candidate to answer this question. In the next sections, two alternative approaches to ubicomp environment modelling are described: one approach centred on users, and one approach centred on sensors. The next chapter presents in more detail the chosen approach and illustrates it through an example.

3.2.1 User-centred approach

Creating large models can be a complex task. Using CPN the construction of large models can be divided into smaller pieces by using *substitution transitions*. Models with these transitions have multiple levels of detail. A global model can give a broad view of the system using *substitution transitions* that abstract detailed models at a different level. Modules are integrated into the global model using *substitution transitions* represented by double line boxes (see Figure 3.8). See Section 2.2.2 for further information about *substitution transitions*.

The user-centred approach collects the user's position from *OpenSimulator* (*GetData substitution transition*, see Figure 3.8) and uses it to make decisions. The example of a smart library is used where gates are opened when registered users approach them. Depending on the user's position, different *widget* transitions can be enabled. *Widget* transitions provide access to information while hiding details of sensing it. Once executed, these transitions, which react to a user's action or change of context, affect the behaviour of the dynamic objects as represented by transitions 1, 2 and 3 in the module. For instance, when the user is near the entry gate, the *Widget1* transition is executed and consequently the *Screen* and *Door transitions take place* to update the information of the screen and open the door. The *object behaviour* transitions (e.g. Screen, Door and Book) describe the behaviour of the dynamic objects present in the environment. Figure 3.9 presents the model of the Screen. This is abstractly represented in Figure 3.8 (tag 1). The state of the screen and the information displayed change depending on the *widget* transition satisfied (user near or far from the entry gate). At the end of the execution of this model a token is put on the *and* place and consequently the *Door* module is executed (see Figure 3.8, tag 2).

The approach presented describes one possible way in which CPN can be used to model ubiquitous environments in a way that makes possible the simulations that are of interest. However, in terms of the scaling of dynamic objects and *widget* transitions there are limitations. For example, to add another screen to the environment, a new screen module must be added (a clone of the presented screen module, see Figure 3.9). To deal with additional situations not present in the current model (e.g. a different user's action or context change) new *widget* modules must be added. For the modelling of ubiquitous environments, with many devices, this limitation makes this approach impractical. Indeed, this approach does not scale easily.

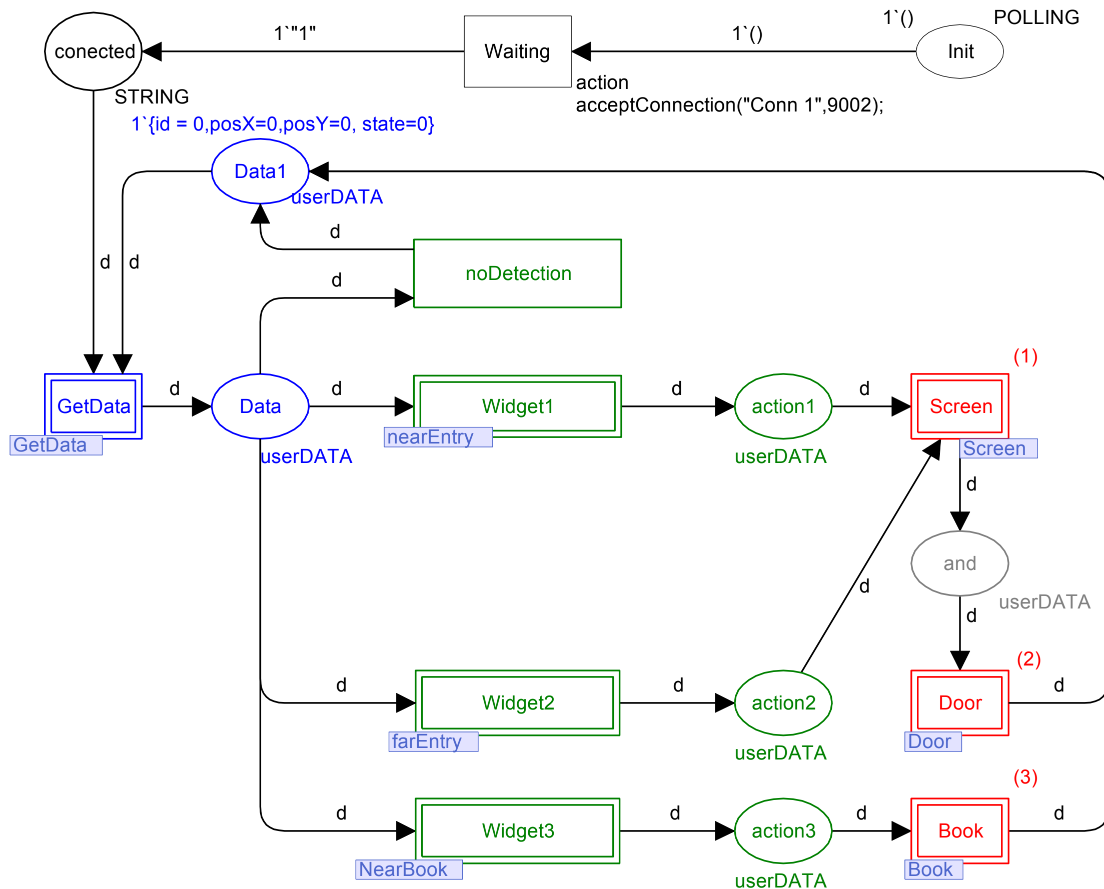

Figure 3.8: User-centred smart library module

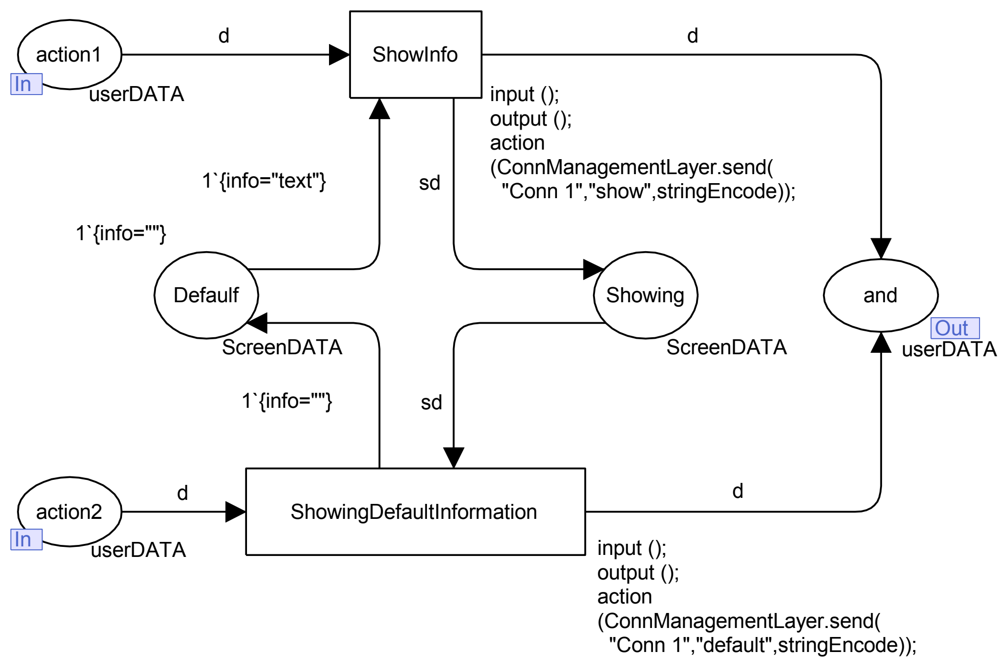

Figure 3.9: User-centred screen module

Approach's improvements

Improvements were made to the previous approach to remove scaling problems. Each type of element (e.g. gates, screens) is represented by a *fusion* place (see Section 2.2.2 for further information about *fusion* places) that holds a token for each element (dynamic object or user) of that type. For instance, the *fusion* place *gates* (Figure 3.10) holds a token for each gate present in the environment. Every type of element present in the environment is added as a place to the initialization module (Figure 3.10) and elements of each type as tokens in their respective places.

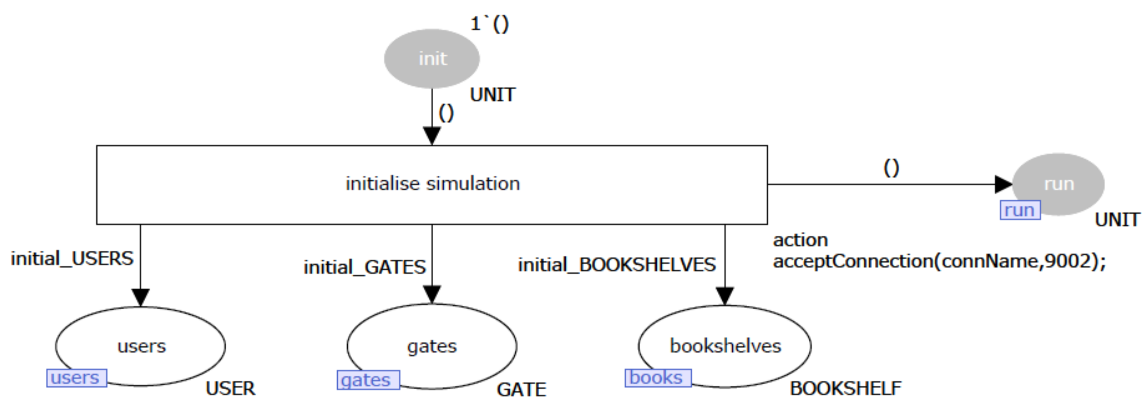

Figure 3.10: General early initialization module

In comparison with the previous approach another significant improvement is that instead of describing the behaviour of the dynamic objects with *substitution* transitions this is now modelled using *fusion* places in a new module (see the *users* and *gates fusion place* of the gate module in Figure 3.11). This reduces the modelling complexity, making it more flexible and practical. For example, to add a new screen it is enough to add an additional screen token to the model. This also has the potential to support the dynamic addition and removal of objects, users and sensors at runtime.

The module presented in Figure 3.11 expresses the behaviour of the gates (opening and closing) using this new approach. The notion of *widget* transitions is now implicit in the modules, see for example, the condition *isArrivingToGateArea(u,g)* in transitions *show info and open gate*. This function determines when a user *u* approaching the gate *g* is considered to have arrived in the gate area (in this case when the distance between the user and the gate is lower than 2). The behaviour of this function described in the ML language is presented in

Figure 3.12. This module leaves a gate open while at least one user is close to it and closes it when no one is near to it.

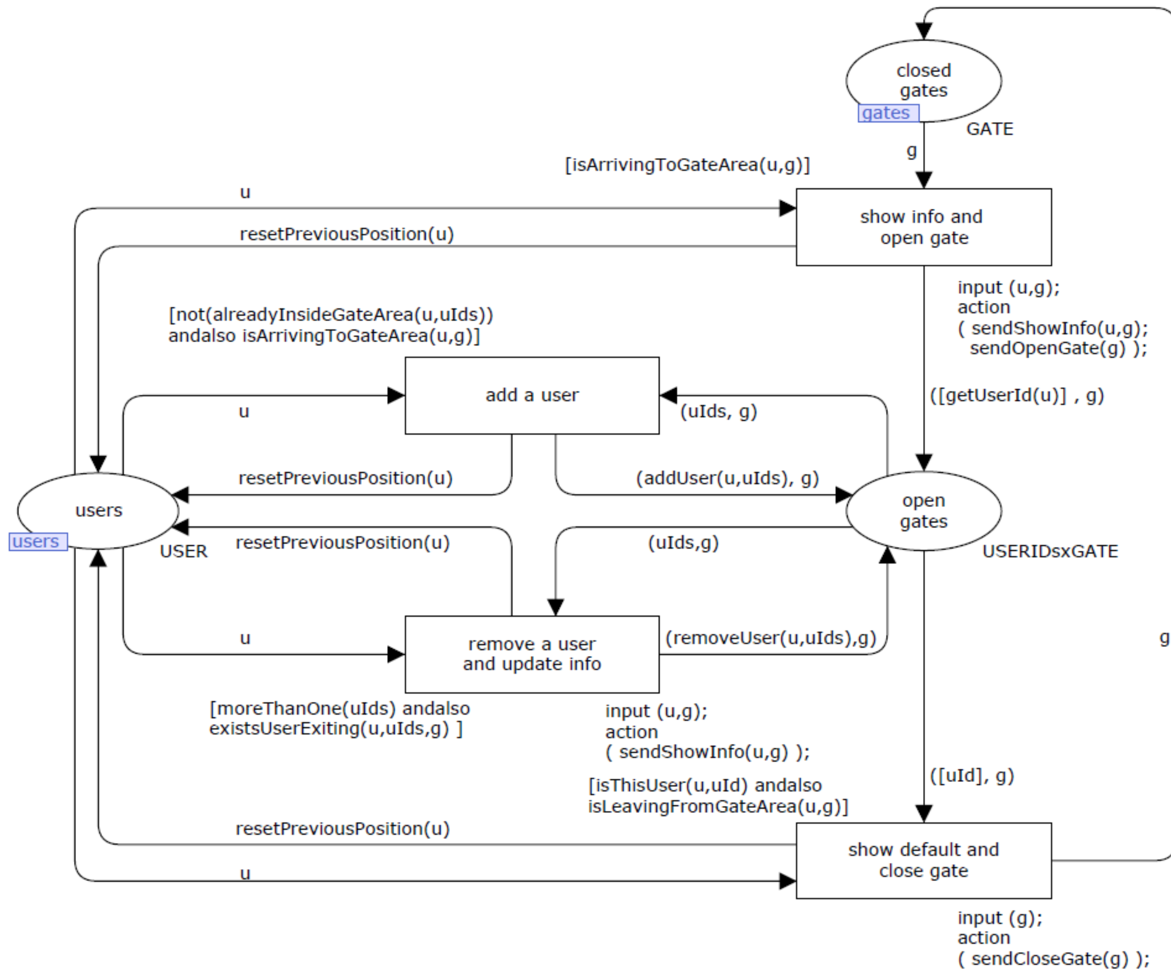

Figure 3.11: Module for an entry gate

```
fun isArrivingToGateArea(u:USER,g:GATE) =
let val d = Math.sqrt(
Math.pow(Real.fromInt(getPosXGate(g)-getPosXUser(u)),2.0) +
Math.pow(Real.fromInt(getPosYGate(g)-getPosYUser(u)),2.0) +
Math.pow(Real.fromInt(getPosZGate(g)-getPosZUser(u)),2.0))
in ( d <= 2.0)
end;
```

Figure 3.12: *isArrivingToGateArea* function

This model collects the user’s position directly from OpenSimulator. Figure 3.13 presents the CPN module that collects the users' data. Transition *read user id* reads a user's identifier. A token associated with the value of the user identifier is introduced in the place *read user id*. This is used to read the new position by means of the transition *read and update user position*, which also updates the relevant user token. The expression, *isThisUser(u,uId)* in the guard of this transition, guarantees that the user token that is updated corresponds to the previously read identifier.

CPN modules for reading each user's position and describing the devices' behaviour execute concurrently. The concurrency issue associated with CPN is that more than one transition can, at the same time, be enabled to execute. Different transition execution orders can lead to different results. These situations must be avoided. Therefore the precedence of some transitions over others must be defined. Precedence of the devices' transitions over the data acquisition transitions is guaranteed through the *not (hadASignificantMovement(u))* guard on the transition *read and update user position* in Figure 3.13. Movement of a user is significant (for a device) when the new position is *near* the device. The importance of this precedence is related to the fact that all device transitions must be executed before a new user data acquisition happens. Otherwise, transitions can be disabled according to the new acquired values rather than executed with the past user values leading to the possibility of losing some behaviours. For example, the transition to open a gate can be enabled at a certain time because a user is near to it. However, if no transition precedence is established in the model, a new user position value can be acquired (before the *open gate* transition execution) which will disable the transition to open the gate. In this situation the gate should open but it does not and therefore behaviour is lost. In the most recent version of the CPN Tools (since version 3.0) prioritized transitions are supported. This means that the precedence of a transition over the others is now easily achievable by simply specifying its priority. Figure 3.14 show three transitions with associated priorities. When both are enabled at same time transition *T1* executes first then transition *T2* and finally transition *T3*. This is because transition *T1* has the highest priority (*P_HIGH*) and transition *T3* has the lowest priority (*P_LOW*). The transition T2 possesses normal priority (normal priority is hidden by CPN Tools to improve a model's readability). Other intermediate priorities can be specified.

In APEX, transition priorities are mainly used to guarantee that all enabled transitions execute before the transition that is responsible for reading new information from the *simula-*

tion component (OpenSimulator). To avoid this situation the reading of new values has lower priority than the processing of values already read. Some of these situations are present in modules of the CPN *base model* (Appendix A).

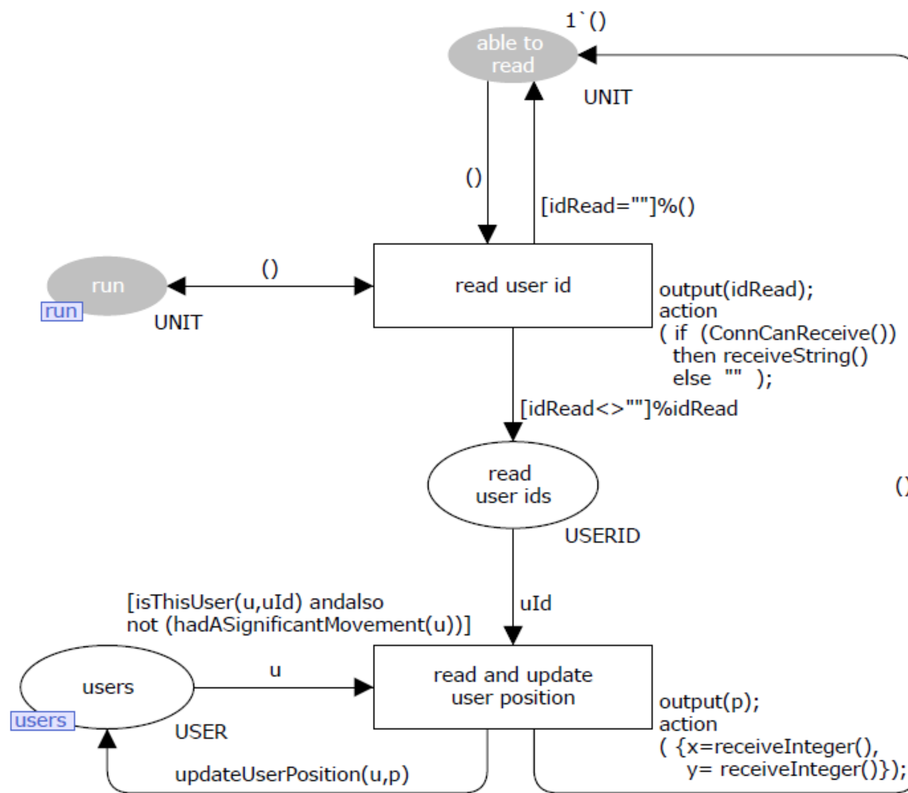

Figure 3.13: Generic module for acquiring users' data

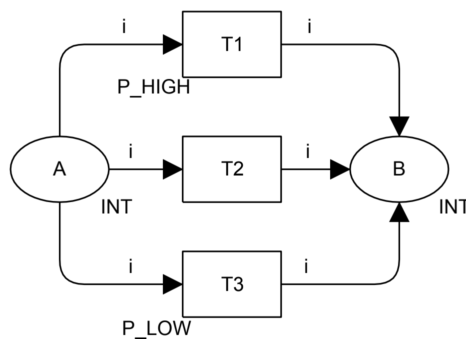

Figure 3.14: Priorities over transitions

Besides the improvements introduced as discussed above this approach has a remaining major limitation that is related to the fact that it is user-centred i.e., it is based on the assump-

tion that the user's positions are always known every time and everywhere. This is not always realistic in the sense that in a real implementation the user's positions are not readily available. Sensors are needed. The position of a user is only known when "near" a presence sensor. One implication in terms of modelling is that this new approach could be more flexible to model context (sensor information, conditions, etc.). The context is currently mixed with the dynamic object modules. This prototyping approach could therefore be more modular because when conditions or sensors change this implies changes in the dynamic object behaviour modules. The approach should separate the sensors' data acquisition models and the dynamic object modules to achieve a more modular approach.

This approach does not entirely satisfy some of the enumerated criteria presented above (e.g. realistic, flexible and modular). The second approach to modelling, presented in the next section, better satisfies the enumerated criteria.

3.2.2 Sensor-centred approach

The approach that was eventually chosen avoids some of the limitations of the previous one, focusing the model more on sensors and less on users. For example, the users' positions are now obtained from the positions of the sensors used to detect users (*presence sensors*). Figure 3.15 illustrates the approach. The revised models are simpler than the previous ones. They are required to:

1. read sensor information from each sensor type;
2. create/delete/update respective tokens.

One model for each sensor type is required because each sensor type has different features (e.g. *movement sensor*, *light sensor*). These models collect the information that comes from the simulation, creating a token and updating their attributes (values). After that, the token is placed in the *fusion* place of the respective sensor type. A *fusion* place for each sensor type is present in the model.

As a result of this, the *fusion places,* holding sensor tokens (places *PresenceSensors* and *LightSensors* in Figure 3.15 part a and b), can be used in the specification of the behaviour of the environment (Figure 3.15 part c). For example, the model described at the bottom of the Figure 3.15 (part c) executes an action depending on satisfaction of a condition *condition(ps, ls, obj)* that depends on the token values present in the *PresenceSensor, LightSensor* and *State*

A places. The *presence sensor* and *light sensor* simplified models of the figure represent the models that collect the information coming from the simulation.

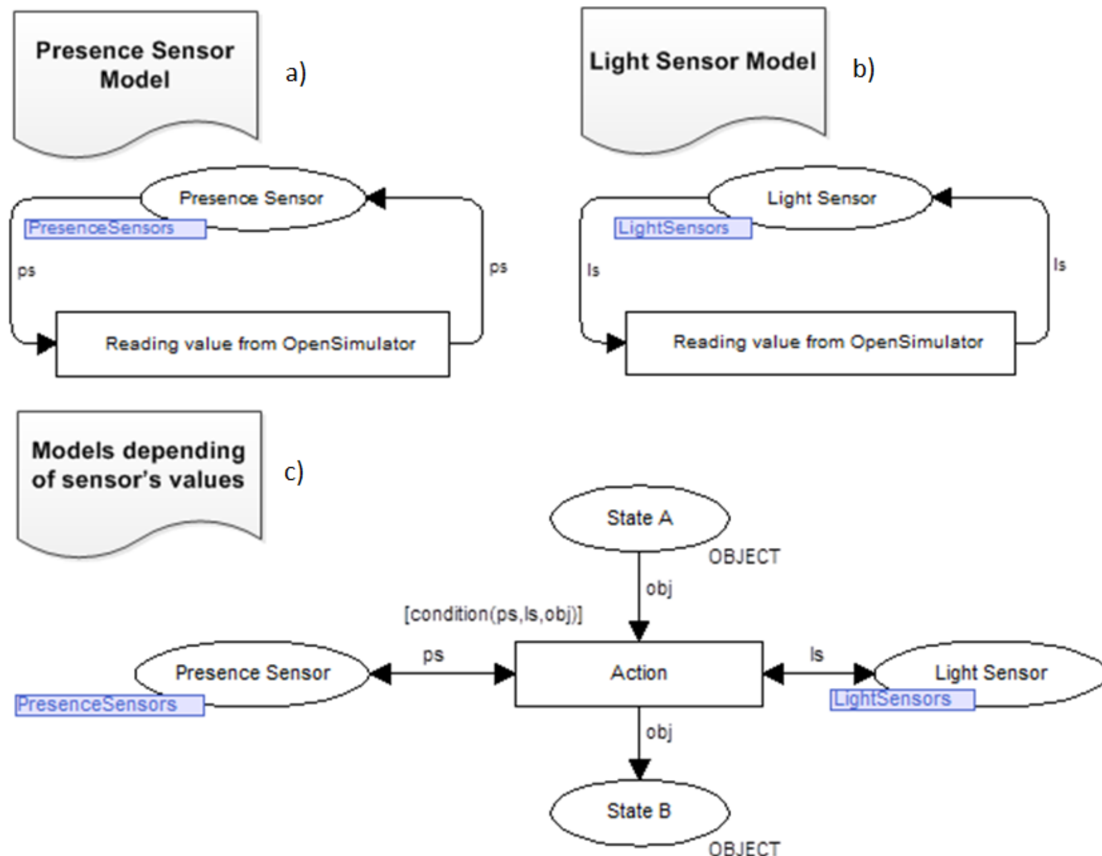

Figure 3.15: APEX sensor-based modelling approach. Parts: a) Presence Sensor Model b) Light Sensor Model c) Models depending of sensor's values

The sensor models that acquire sensor information are a part of the CPN *base model* and hold the information read in *fusion* places (e.g. *PresenceSensor fusion* place), see Figure 3.15. The models describing the environment elements (e.g. gate) are added by developers and linked to the *base* model, using the information collected by the sensor models, as shown in part c of Figure 3.15.

This approach has advantages when compared with the previous one. It is more realistic because it reflects more accurately the behaviour of ubiquitous environments. The approach also separates the data acquisition modules for the sensors from the dynamic object modules. This makes the model more modular and eases scaling. Both approaches are generic because they can be instantiated to different situations. The sensor-centred modelling approach is

evaluated with different examples in Chapter 7 thus illustrating how generic it is. This modelling approach is the one that has been selected and is described in more detail in the next chapter.

3.3 Conclusions

The aim of rapid prototyping frameworks should be to make complex system development easier and more efficient. To achieve this, these frameworks should be flexible and extensible. The APEX framework focuses on the prototyping of ubiquitous computing environments. Its architecture supports different types of sensor and allows their introduction to be achieved incrementally within different layers. APEX contains components that are responsible for separating prototyping aspects (behaviour, simulation and physical characteristics) and a component responsible for the management and synchronization of the information flowing between them.

The architecture provides layers in which a prototype design can be evaluated. Different layers enable exploration of how the user would experience the proposed design in one layer while enabling an exhaustive analysis of the design implications in another layer. The framework also supports a development process in which virtual, physical or mixed elements are explored depending on the availability of these components. The initial stages of development can be achieved entirely in terms of CPN models. Further development can be moved into the virtual world before moving wholly or partially into the physical world. This supports exploration of how different levels of abstraction can be accomplished and supported. For example, supporting and enabling the migration of devices at the physical level via Bluetooth, at the virtual level as virtual devices in OpenSimulator, at the model level as CPN models. In summary it is possible to explore the design from a variety of perspectives.

During the development of the framework, and according to its architecture, a new C#/CPN communication package has been developed enabling the communication between CPN models and C# processes.

The modelling issue, according to the requirements, is to support a combination of realism and tractability. Two modelling approaches were presented, one user-centred, another sensor-centred. The sensor-centred approach provides a more realistic simulation, closer to what occurs in real ubiquitous environments. It is modular, scaling to different types of sen-

sors (presence, luminance, etc.) and providing an incremental style of prototyping. It is presented in more detail in the next chapter.

Chapter 4

The Modelling Approach

This chapter describes the selected modelling approach. It starts by describing how to set up a model simulation. Subsequently, the CPN *base model* that forms the basis of this modelling approach is explained by presenting its structure, its logic and how it can be extended to different examples. The chapter continues by describing the modelling and use of programmed avatars. Finally, conclusions are presented.

4.1 Using CPN to generate a simulation

The initial conditions of the simulation are defined in the CPN initialization module shown in Figure 4.1. Firing the *initialize animation* transition sets the default configuration of the simulation, and executes the associated CPN ML code. In this particular case the configuration includes seven places (users, PDAs, dynamic objects and four types of sensors). The *users* place holds *USER* tokens representing information about users in the virtual environments (avatars). This particular place is mandatory, since whatever the model the handling of users must be supported. The remaining places hold tokens representing devices and sensors. All these places are environment dependent and will vary for each prototype. The *colour* (structure) of the tokens that these places can hold is defined in CPN Tools, and characterises the information held in the model for each type of device. The initialization module uses two places to control the initialization of the CPN model: *init* to limit execution of the transition to one occurrence, and *run* to inform other modules that the simulation is running. Once all desired places and modules are added to the model the simulation can start. The CPN Tools provides support for model simulation (e.g. *play*, *stop*) that is similar to program execution.

The detailed setup (modelling and prototyping) and use of APEX is explained in Appendix E.

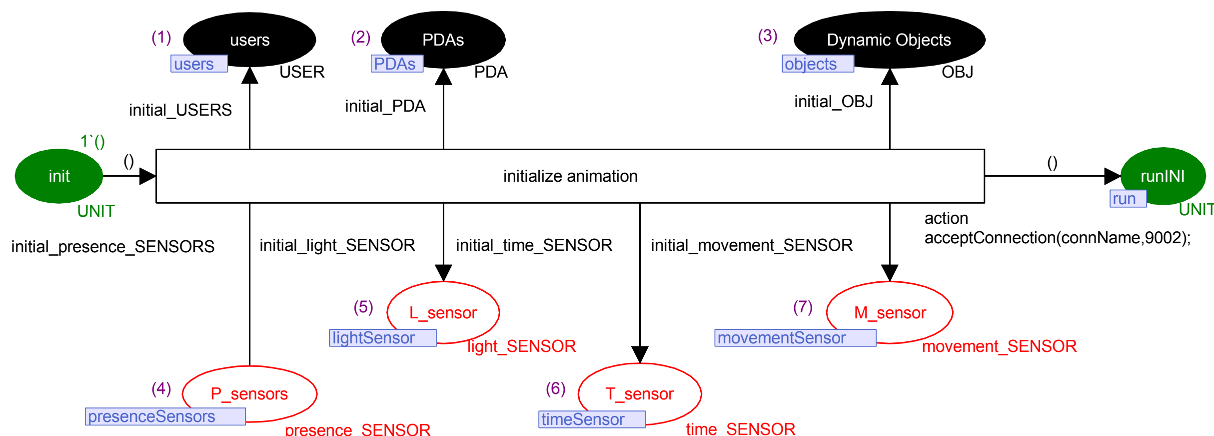

Figure 4.1: General early initialization module

4.2 The CPN Base Model

In this section the modules that form the CPN *base model* are described. The use of the *base model* in modelling new situations is then described. The extension of the *base* model is presented in the following sections. The description shows:

1. how to setup of the CPN *base model*, presenting the environment's devices modules and the connection between the model and the OpenSimulator server (sub-Section 4.2.1);
2. the modeller's tasks that must be carried out to use this modelling approach in new situations (sub-Section 4.2.2);
3. the detailed modelling of the environment's device modules (sub-Section 4.2.3).

4.2.1 The model

The CPN *base model* as described in the previous chapter needs to be extended by developers to respond to changes in the environment (user's actions or context changes) including forwarding model state changes to the environment.

Setup

The *base model* needs to be set up before being used in new situations. Many of the modules of the CPN *base model* are generic and do not need any modification. The developer only needs to develop a module for each type of device present in the ubiquitous environment, and only if the desired module does not yet exist (previously developed modules can be reused), and to add them to the *base model*. The connection is established by means of *fusion* places (see Section 2.2.2).

Figure 4.1 describes the *initialization* module, presenting an overview of all possible elements (dynamic objects, sensors and users) present in the simulation. This module is an improvement of the early user-centred initialization module presented in Figure 3.10. This sensor-centred module is responsible for initialising the elements with *default* values and for establishing the connection to OpenSimulator. The connection is described later in this sub-section.

The *fusion* places in the top part of Figure 4.1 (annotated with the numbers 1, 2 and 3) hold information about dynamic elements (*users*, *PDAs* and *Dynamic Objects*). The *Dynamic Object* place (annotated with number 3) contains one token for each dynamic object (e.g. gate, screen) present in the environment except for the mobile devices that are held in the *PDA* place because these object can receive information from the users (e.g. text) being dealt differently by the model. Additionally, the data type of the *PDA* place is different from the *Dynamic Object* place. Each *dynamic object* token contains position, identifier and type (whether the object is a book, a gate, etc.). Objects such as walls are considered to be *static objects* and do not need any associated behaviour (they are just present in the *virtual environment component*). The *users* place (number 1) holds a token for each user connected to the simulation. These tokens hold information such as users' identifier and users' information (e.g. users' requests). The *fusion* places at the bottom (annotated with the numbers 4, 5, 6 and 7) hold the sensor tokens. Each place holds the tokens of a specific sensor type (e.g. *presence* sensor). Because they are *fusion* places their value is accessible anywhere in the model.

The model is automatically initialized with the elements that compose the environment, however initial values can be defined modifying the initial variables (e.g. *initial_PDA*).

Environment's devices modules

The environment's device modules are presented in the context of the smart library example. The main feature of this library is to provide users with directions that they should follow to reach desired books (describe in more detail in Section 7.1). Figure 4.2 and Figure 4.3 present modules of two specific types of devices that are present in this example (gates and book lights) used in extending the *base model* to obtain the expected behaviour of this ubiquitous environment.

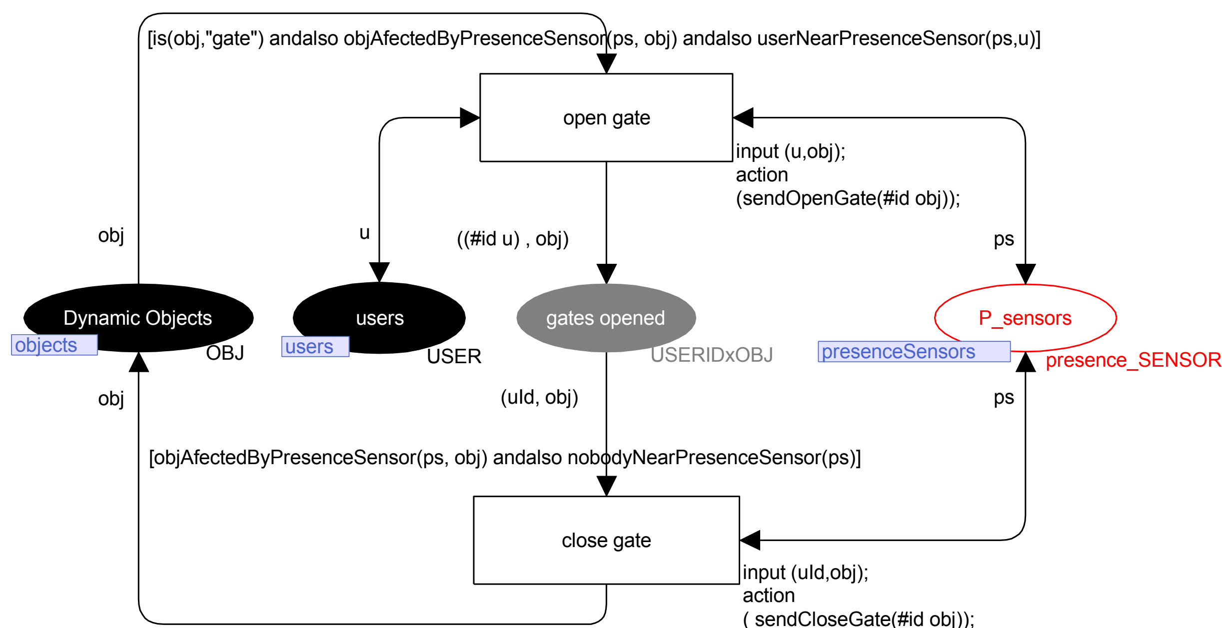

Figure 4.2: Gate module

Fusion places are the basis for the creation of these behavioural modules and enable token flow between them. They establish the link between the devices' modules, the initialization module (Figure 4.1) and all other relevant modules where the information is needed. For example, the device module of Figure 4.2 is connected to the initialization module via the *users*, *objects* and *presenceSensors fusion* places. The connection between the model and the simulation is accomplished through functions associated to state transitions (e.g. *sendCloseGate*). Functions are also responsible for describing functional behaviour not structurally expressed by the net.

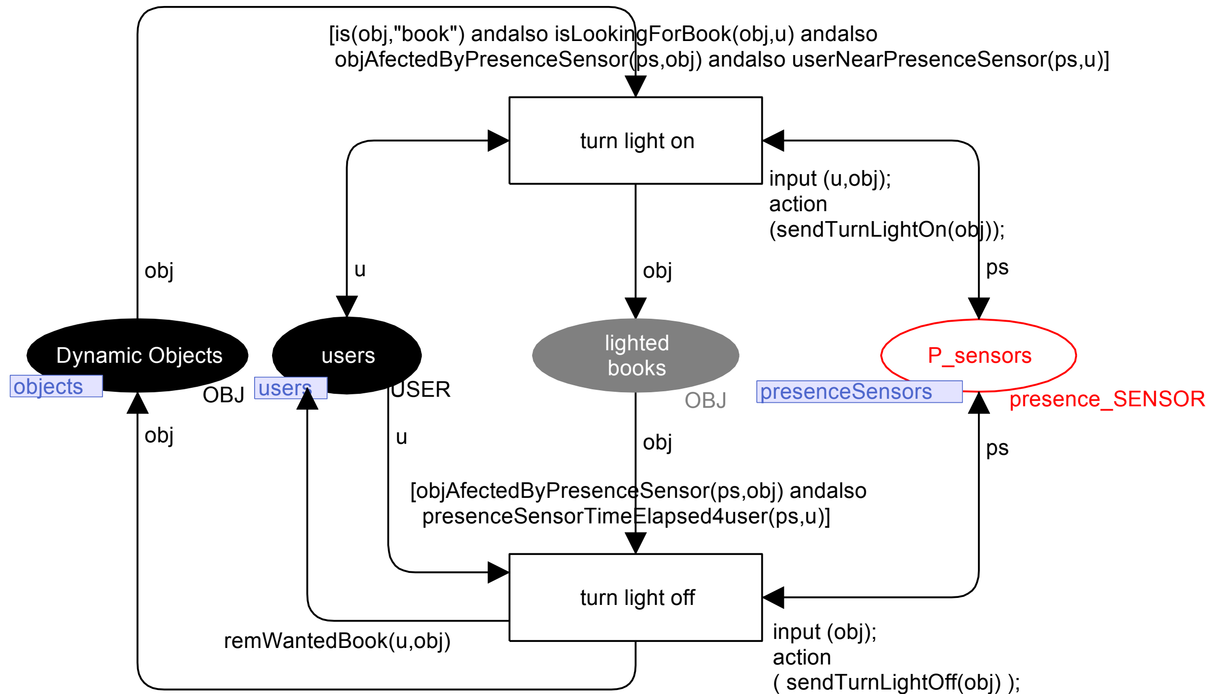

Figure 4.3: Book's light module

Figure 4.2 shows a module describing the behaviour of gates following the sensor-centred approach. The behaviour of the gates is modelled making use of the values acquired from the sensors. Gates can be in two states, opened or closed. The gates should open when a user is near to the presence sensor associated with it and be closed when nobody is close to it. The module holds the gates that are open in the *gates opened* place, and in the *Dynamic Objects* place the gates that are closed (see Figure 4.2). The *open gate* and *close gate* transitions move tokens between states. *Fusion* places (*objects*, *users* and *presenceSensors*) are used in the modelling, providing access to values required for the specification of conditions associated with the transitions (e.g. *is, objAffectedByPresenceSensors*) that outline the desired behaviour. These places provide access to the tokens that animate the module.

When devices are present in the environment for which a CPN module is not available, a model of the behaviour of the device must be developed, following the same reasoning as above. Figure 4.3 shows the book light module for the smart library that turns on and off the book lights that users have requested when they approach/move away from them. Using the user's position, the position of the desired books and, the user's desired books list, the module decides to turn on some books' lights. This information is then forwarded to the virtual environment (*sendTurnLightOn* function of the *turn light on* transition). This information is inter-

preted by the APEX *comunication/execution* component and is provided to the books so the avatars are aware that the light is turned on or off. When the avatar stops being detected by the presence sensor responsible for the trigger of a book light (*presenceSensorTimeElapsed4user* function of the *turn light off* transition) the light is turned off and the book removed from the user's list (*remWantedBook* function).

The number of tokens present in the model may vary during the execution depending on what is present in the environment. Users can connect and disconnect, devices/sensors can be added, deleted or updated. These modules reflect the changes of the environment. The modelling of devices is presented in more detail in Section 4.2.3.

Communication between the model and the 3D simulation

The connection and communication between the model and OpenSimulator is now described. The execution of the *initialize animation* transition of the initialization module (Figure 4.1) means that the connection between OpenSimulator and CPN Tools is successfully established and devices are initialized with default variables' values (e.g. *initial_USERS*). The default variables present in this module hold the values with which the tokens representing the devices must be initialized at the beginning of the simulation. Most times the devices are initialized with empty values because they are then updated reflecting the environment state.

The binding between OpenSimulator and the model is accomplished by both the *communication/execution* component and the functions of the Comms/CPN library in the model. For example, the *acceptConnection* function present in Figure 4.1 allows external processes to connect to the CPN Tools. The identifier (*connName*) and the port (*9002*) of the connection are provided as arguments and are the same as the ones used by the *communication/execution* component. This component can be loaded by the OpenSimulator server as a DLL and consequently can access all elements of the environment and being able to forward the acquired information (from the model) to the environment.

Having described the connection between the CPN model and OpenSimulator, the exchange of information between the device modules and OpenSimulator is now addressed. Part of the information exchange binding between the model and OpenSimulator is present in the transition's *action* parts of the model (see, Section 2.2.2) which are responsible for modifications to environment objects and are triggered during a transition's execution. This is also illustrated in the device module present in the Figure 4.2. The *action* associated with the *open*

gate transition is the *sendOpenGate* function. This function is responsible for sending two things to the *communication/execution* component, the *id* of the gate to open *(#id obj*) and an indication to open it (information implicit in the *sendOpenGate* function). The code of the functions used is described below. The argument of the function *sendString* is the string containing the identifier of the gate to open (*gId*) and the indication to open it (*exe*). This information is interpreted by the component and results in the update of the virtual environment according to the information sent (this process has been described in Section 3.1.3). The *sendString* function uses the *send* function from the Comms/CPN library to send information to CPN Tools external processes (in this case the *communication/ execution* component processes) as illustrated.

```
fun sendOpenGate(gId:GATEID) = sendString(gId^"-"^"exe");

fun sendString(s: STRING) = ConnManagementLayer.send(connName,s,stringEncode);
```

4.2.2 Modeller's tasks

The tasks that the modeller carries out in the approach are now presented. The focus of this section is to describe the tasks that precedes device modelling and that must be performed. Firstly, all the types of *dynamic objects* that are needed (present in the environment being prototyped) must be identified. In the illustrated example, *books*, screen and *gates* are the *dynamic objects* needed (*users, PDAs* and *sensors* are not considered as *dynamic objects*). For each object type the following steps must be developed:

1. *Colour sets* must be created by specifying their data structure (e.g. gate's *colour set: colset GATE= record id: GATEID * position: Pos3D;*), if they do not yet exist, as illustrated in Figure 4.4;
2. Clones of the existing *fusion* places needed must be created (e.g. *dynamic objects*, *users* and *sensors*) that hold the tokens of the elements needed to specify the behaviour of the device modules;

A module must be created which describes the behaviour of *dynamic objects* of the same type (device module) if it does not yet exist. For that purpose *fusion* places together with functions are used. The provided gate module (see Figure 4.2) is one of these device modules. The next section focuses on the description of the modelling of device modules.

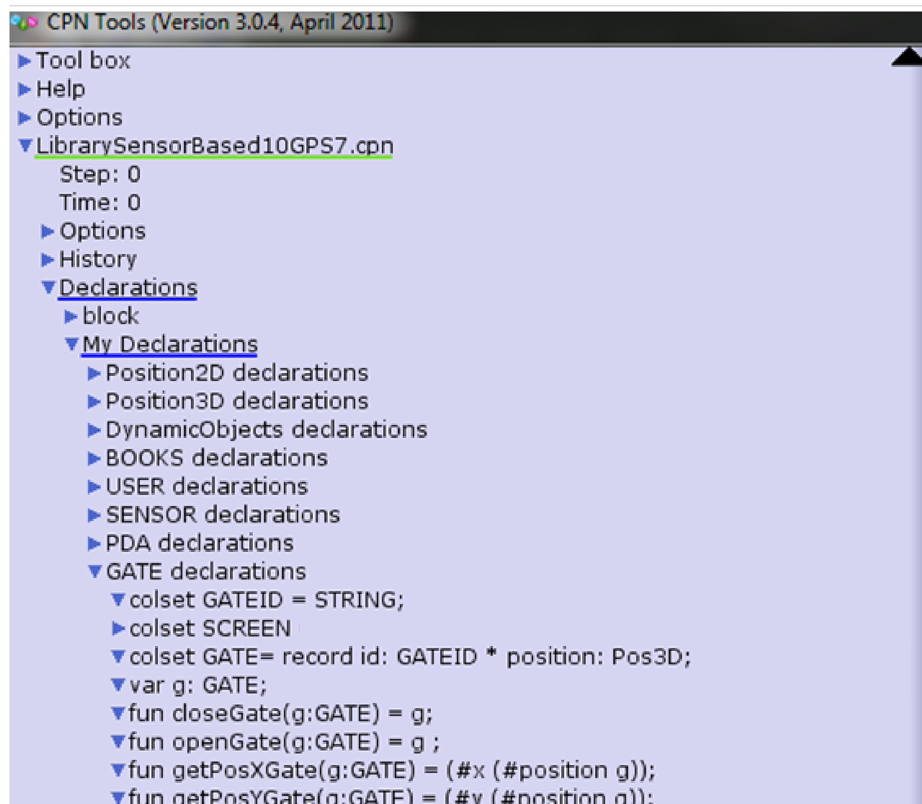

Figure 4.4: *Colour set* declaration (library example)

4.2.3 Modelling environment's devices

Each type of dynamic object in the ubicomp environment simulation needs a corresponding CPN module describing its behaviour (device module). A library of modules is available for supported devices. When new (unsupported) devices are to be used, a new module must be developed and added to the *base model*. This section explains the process of modelling these devices using the gate in the library example (Figure 4.2).

The devices' behaviour is modelled through a combination of *fusion* places, normal places, transitions, functions and conditions. The places of the gate module (*users*, *Dynamic Objects* and *P_Sensors)* hold the tokens of the users, gates and presence sensors needed to model the behaviour. The gate is equipped with a sensor to capture a user approaching it. The *open gate* transition represents the actions relating to the gate, for example it opens the gate. Function *sendOpenGate* is responsible for this action, sending relevant instructions to be reflected in the environment. This action occurs when the gate's sensor detects a registered user arriving at the gate (modelled by *userNearPresenceSensor(ps,u)* evaluating to true) and this

sensor effects a transformation on the gate (*objectAffectedByPresenceSensor(ps,obj)* evaluating to true). When the gate is still open and another registered user enters the gate area the transition *open gate* does not occur but the user can enter.

As already stated, each place holds an associated token type. In this case the type of the gates opened place is *USERIDsxOBJ*. Each token is a product of a colour set of user identifiers (*USERIDs*) and the gate (*OBJ*) which opened in response to a user. The type *OBJ* represents any dynamic object type. In this situation gates are held in the places of the *OBJ* type. When the transition *close gate* is taken the gate is closed. For this transition to occur users must have moved away from the gate.

The ML code of the *userNearPresenceSensor* and *objAfectedByPresenceSensor* functions associated to the transition's conditions are listed below:

```
fun userNearPresenceSensor(ps:presence_SENSOR,u:USER) =

(mem (#values ps) (#id u));

fun objAfectedByPresenceSensor(ps:presence_SENSOR, obj:OBJ) =

mem (#objectIDs ps) (#id obj);
```

To determine that a user (*u)* is near a presence sensor (*ps)* the *userNearPresenceSensor* function checks if the user identifier (*#id u*) is present (*mem* function) in the set of the presence sensor detected users *(# values ps*). Whenever a user gets near to a presence sensor its set of identified users is automatically updated. This is accomplished by both the *communication/execution* component and modules of the *base model* developed for this purpose. See for example the *presenceSensorsUpdate* module in the Appendix A.

The *objAfectedByPresenceSensor* function behaviour follows the same reasoning. In this case the values consulted are the sets of object identifiers that this presence sensor transforms *(#objectIDs ps*).

4.3 Modelling and Use of Programmed Avatars

In the example the avatar's behaviour is not modelled and, as already discussed, avatars are controlled by real users through the viewer. This allows human users to experience the devel-

oped ubiquitous environment immersively. An alternative possibility is for an “out of the box” view of the system to be achieved through the use of programmed avatars. A module is developed that sends the avatar’s behaviour to the simulation as illustrated in Figure 4.5. Some assumptions are made about programmed avatar's behaviour, only avatar movement is considered. The modelling of programmed avatar interactions is currently not supported. For each avatar a path is defined in the model. This path is represented by a list of positions (e.g. *[{x=126,y=126} ,{x=126,y=128}, {x=120,y=120}]*). The Z coordinate is automatically obtained from the environment based on X and Y coordinates. The programmed avatar module is responsible for sending the positions that the avatar must reach. In the action part of the *send path to user* transition the next position is sent to the simulation (*sendUserPosition(u)*) and then removed from the user’s list (*remLastUserPosition(u)*). In this module a time sensor was used (*fusion* place *timeSensor*) to control the time that the avatar waits before moving to another position. Each time an avatar moves, the time of the movement is saved in the *history* place. The control of the instant to trigger a movement start is now possible with this additional information. The condition *modeAUTO(u) andalso morePos(u) andalso elapsedSeconds(ts,ts1,10)* restricts the transition execution (programmed avatar's module) to when the avatar is in programmed mode (*modeAUTO*), there are more avatar's positions to read (*morePos*) and the delay before the next transmission has occurred (*elapsedSeconds*), in this case 10 seconds.

4.4 Conclusions

The focus of this chapter has been to describe the selected modelling approach. The CPN *base model* was introduced including how to extend it to in order to create the desired behaviour was described. The connection between the model and the virtual environment was also presented.

The base model consists of an *initialization* module and devices modules. In addition to these, modules are provided that are responsible for information exchange between models and the environment, ensuring mutual updating and synchronisation. None of these modules (see, Appendix A) need modifications in the application of this approach to a new situation. However, new modules are required for new devices (defined in *dynamic objects*) present in the environment. They will be required to model different behaviour than the developed ones.

The work required to develop these modules is analogous to the gate module used as an illustration. *Fusion places* representing users, sensors or dynamic objects within the environment are used to model the desired behaviour. The definition of additional functions for specific situations might also be needed.

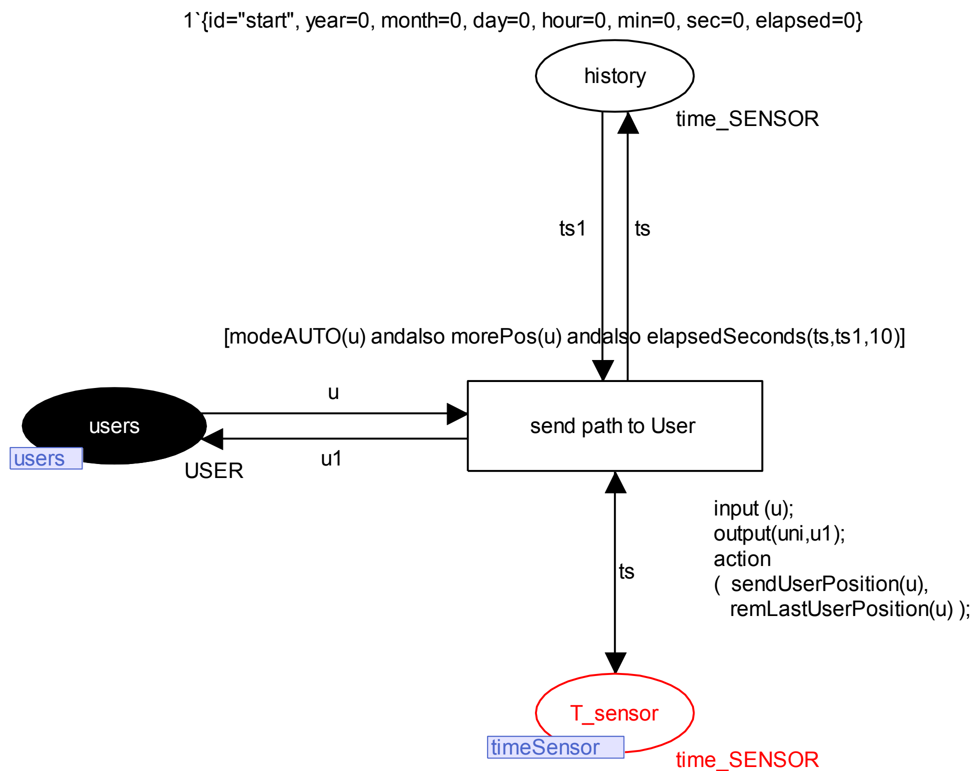

Figure 4.5: Programmed avatar's module

The main sensor types were developed (presence, light, movement and time sensors) but others may be required. Other useful sensor types can be easily developed and simulated in the OpenSimulator (e.g. accelerometers, orientation sensors, etc.).

The development of programmed avatars, reducing the human resources cost to evaluate an ubiquitous environment, was also presented. The improvement of programmed avatar's interaction (beyond avatar's movement) is planned as future work (e.g. avatar taking objects).

A virtual environment to match the proposed model must also be developed. This is done through the virtual world viewer and is described in the next chapter. Once both models and virtual world simulations for all devices are in place, exploration of the envisaged ubicomp environment can start.

Chapter 5

Prototyping Experience Alternatives

The prototyping of ubicomp environments becomes more complete when, besides models that specify behaviour, we prototype rich enough environments to give to users the sensation of experiencing the final physical environment. This richness, combined with the connection of physical devices, immerses users in the prototype. The social impact of design changes, their usability implications and the effects on user experience, are more perceptible in rich and immersive environments. The research question introduced in Chapter 1 "*can ubicomp environments prototypes address features with the potential to assess user experience without physical deployment?*" is explored in this chapter. Different user experiences are possible and provided through the use of APEX.

User *experience* refers normally to the emotional response persons have when doing something or being somewhere. User *experience* can be defined as “a person’s perceptions and responses that result from the use and/or anticipated use of a product, system or service” [80]. Emotional user characteristics such as mood, personality, disposition and motivation are aspects that might influence user *experience*. In prototyping ubicomp environments APEX provides user (virtual) *experience* to their users without physical deployment. Different types of user *experience* acquirable and exploitable with APEX are presented in the next sections. More information for understanding, scoping, and defining the concept of user experience can be found in the work of Law at al. [81].

5.1 Alternative User's Experiences

3D application servers offer several advantages over simple virtual environments. One particular advantage is the possibility of connection of several clients to the same virtual space. OpenSimulator environments are accessible through a variety of different viewers. Besides the Second Life viewer, a list of third party viewers can be used[33]. The main goal of these viewers is to provide client access to the environment. However these viewers possess different features that provide different experiences for users who use them. Some are developed for specific users (e.g. SL Military), others to support specific visualizations (e.g. stereoscopic 3D visualization) or specific hardware configurations (e.g. multiple display usage). Viewers that provide features intended to improve user experience are presented below. Others, that only improve usability issues associated with the viewer or are concerned with minimizing the performance requirements are not addressed.

5.1.1 Second Life viewer

Since Linden Lab, the company responsible for Second Life™ development, released the code of their Second Life viewer a number of alternative solutions were developed to satisfy different user needs (e.g. stereoscopic 3D visualization, multiple display usage). The Second Life viewer provides all the most basic and important features to enable users to navigate and interact with the virtual environment. With the Second Life viewer spaces can be explored, walking around, meeting people, chatting, shopping, creating objects, etc. Customizing avatar appearance, earning money, creating a business, are other features supported by this viewer. A complete description of functionalities are available at the Second Life website[34]. Figure 5.1 presents Second Life Viewer version 3 in use.

[33] Third party viewers to connect to Second Life or OpenSimulator: http://wiki.secondlife.com/wiki/Alternate_viewers#non-linden (last accessed: 25 February 2012).

[34] Second Life description: http://secondlife.com/whatis/ (last accessed: 15 November 2011).

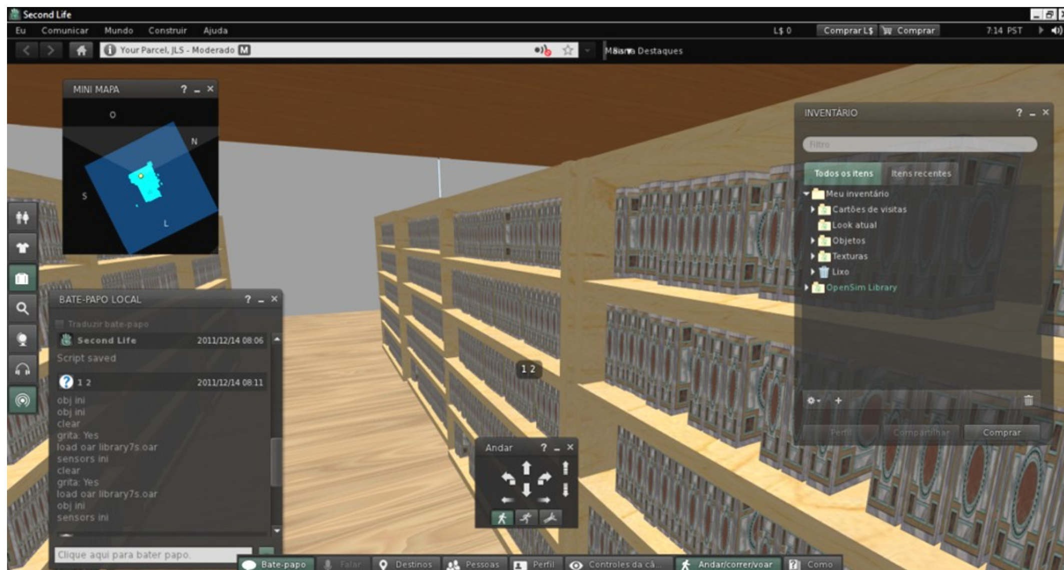

Figure 5.1: Second Life Viewer

5.1.2 Support for importing virtual objects

Virtual objects are typically created through a viewer at runtime. However viewers provide limited support for virtual object creation when compared with game engines. Many thousands of developed objects are freely available including buildings, furniture, daily objects, cars and planes. The Google 3D Warehouse[35] is a popular site for users to share and gather 3D content. Consequently a major advantage for the creation of virtual environments is the possibility of importing virtual objects build using third party tools. The Project Mesh viewer introduces a major advantage over the Second Life Viewer because it is possible to create complex environments just by adding previous or third party developed objects instead of creating all from scratch. The more realistic the prototypes are, the more accurate is the representation of a ubicomp environment, contributing to improving user experience and immersion.

The Project Mesh viewer is based on Second Life viewer. It provides the same functionality but introduces the capability of including objects, known as *meshes,* that are created in external 3D computer graphics software such as Blender, Maya, 3DS Max and Google Sketchup. The file format supported by the objects is the COLLADA 1.4 specification sup-

[35] Google 3D Warehouse: http://sketchup.google.com/3dwarehouse/ (last accessed: 27 February 2012)

porting triangles and polygons (no support for nurbs, surface patches, etc.). This specification format can be consulted at the *khronos* website[36]. Most 3D computer graphics software supports this format and provides tools that enable the conversion from other formats to COLLADA and vice-versa. Once an object is created and saved in the COLLADA format it can be uploaded into the OpenSimulator server using the Project Mesh Viewer's import functionality. The object is processed by the server and represented in the virtual environment. It must be noted, however, that while the Project Mesh viewer is able to correctly interpret these meshes, if another viewer is used (without mesh support), meshes will not be rendered correctly. Figure 5.2 shows the visualization of an environment composed of *mesh* and non-mesh objects through the Project Mesh viewer, and Figure 5.3 shows the result of using a viewer that does not support *meshes*.

Figure 5.2: Rendering of *meshes* with Mesh Project viewer

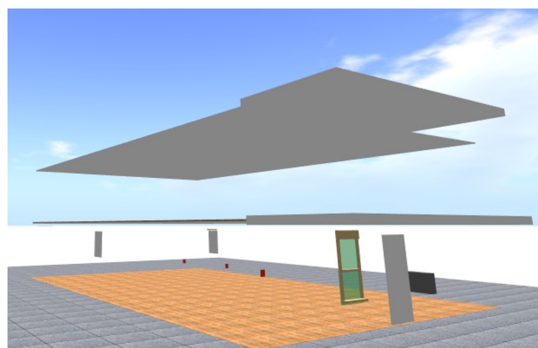

Figure 5.3: Rendering of *meshes* with a viewer which does not support *meshes* (the objects present are not meshes)

[36] COLLADA format specification: http://www.khronos.org/collada/ (last accessed: 1 March 2012)

Since the Mesh Project viewer was made available, Second Life™ showed an interest in the power of *meshes*[37]. Consequently during the summer of 2011 *mesh* support was made available in the Second Life viewer and the Project Mesh viewer was discontinued. Users must now pay for the import of *mesh* objects whereas before it was free. However, more recently the Phoenix Viewer Project Inc. (a non-profit incorporated organization)[38] decided to release a viewer with free *mesh* support, the Firestorm Beta 3 Viewer[39].

5.1.3 Supporting 3D visualization

A different approach to improve user experience, improving the texture of their experience, is to provide users with stereoscopic 3D visualization. Dale's SL viewer improves the Second Life viewer by supporting a set of stereoscopic modes that can be used to improve user immersion. According to Dale's SL viewer web page[40], these modes are:

1. Anaglyph Stereo (Red/Cyan Glasses): the impression of depth is achieved through the use of glasses and two images overlapped with different colours and lagged contours [82] (See Figure 5.4);
2. Passive Stereo (dual Projector): to obtain a stereo effect the use of two projectors for one screen is required and the user requires polarized glasses where each lens only allows visibility of the projected images of one projector [82];
3. Active Stereo (Shutter Glasses): this mode requires shutter glasses and only one projector but with high video frequencies. The stereo effect is achieved by the separation of the frames displayed to each eye. This means that through the shutter glasses the right eye sees half of the frames and the left eye sees the other half [82]. Figure 5.5 shows the process of Active Stereo.

The major advantage of using this viewer is that a stereoscopic 3D visualization providing support to different stereo modes is offered. This visualization provides a richer experience to users because they can feel more immersed. User reaction to design changes can

[37] Second Life™ interest in Meshes: http://community.secondlife.com/t5/Featured-News/Mesh-Coming-to-Second-Life-This-Summer/ba-p/902061 (last accessed: 15 November 2011).

[38] Phoenix Viewer Project Inc: http://www.phoenixviewer.com (last accessed: 6 March 2012)

[39] Firestorm Beta 3 viewer: http://phoenixviewer.blogspot.com/2011/09/firestorm-beta-3-with-mesh-support.html (last accessed: 6 March 2012).

[40] Dale's SL viewer: http://sl.daleglass.net/ (last accessed: 15 November 2011)

therefore be evaluated more accurately because users are closer to the physical ubicomp environment being prototyped. The prototyping gains in realism and therefore user experience is potentially enriched.

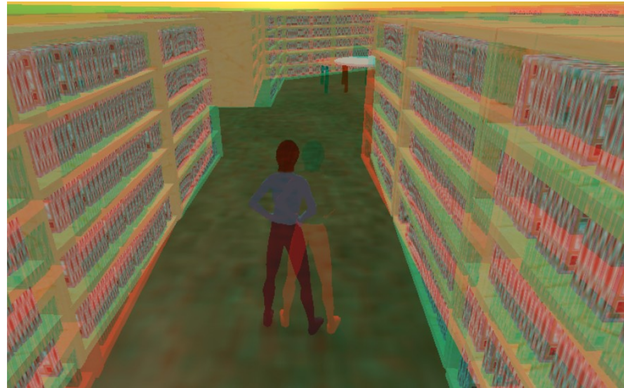

Figure 5.4: Dale's SL Viewer in anaglyph stereo mode

Figure 5.5: Active Stereo - shutter glasses behaviour

5.1.4 Supporting multi displays systems

Another possibility to improve user immersion in the prototypes is to adapt the traditional Second Life viewer for large scale immersive displays such as CAVE (Cave Automatic Virtual Environment, see Figure 5.6) and other multi-projector systems [83]. Multiple computer screens provide immersion that enables a much richer experience than using just one screen. The approach of CaveSL uses multiple concurrent Second Life logins across an unlimited number of machines to provide multiple display support. Figure 5.7 shows data and display synchronization between all viewers (Second Life clients).

The major advantage of this solution is to provide users with a richer and more immersive experience by providing a wide view of the space they are exploring (see Figure 5.8).

Figure 5.6: A CAVE[41]

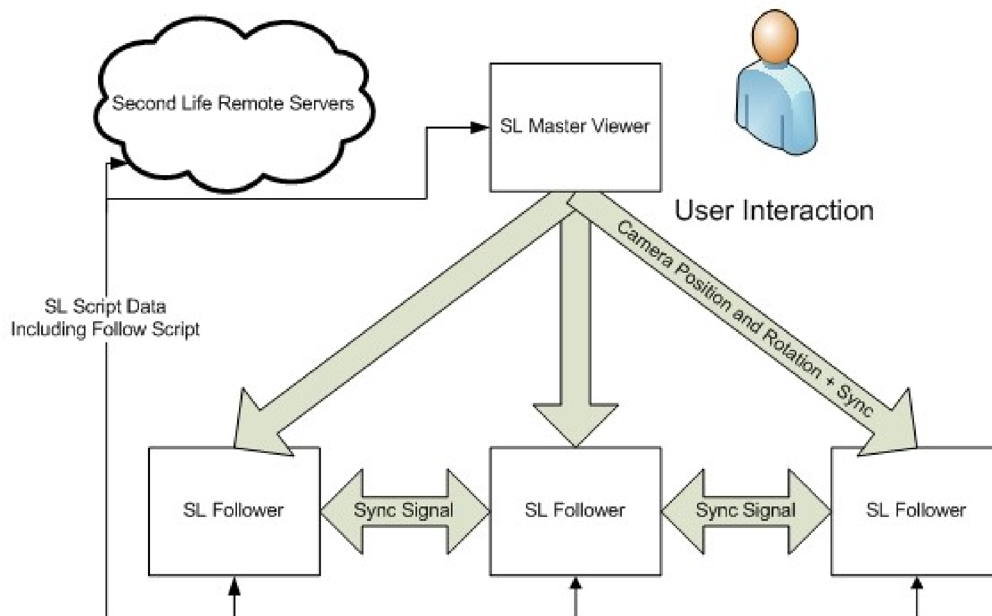

Figure 5.7: Data and display synchronization with CaveSL[42]

[41] Adapted from the Virtual Development and Training Center (VDTC) website: www.vdtc.de/en/en_dru/press-photos.htm (last accessed: 10 January 2012)
[42] Adapted from the CaveSL website: http://projects.ict.usc.edu/force/cominghome/cavesl/index.html (last accessed: 10 January 2012)

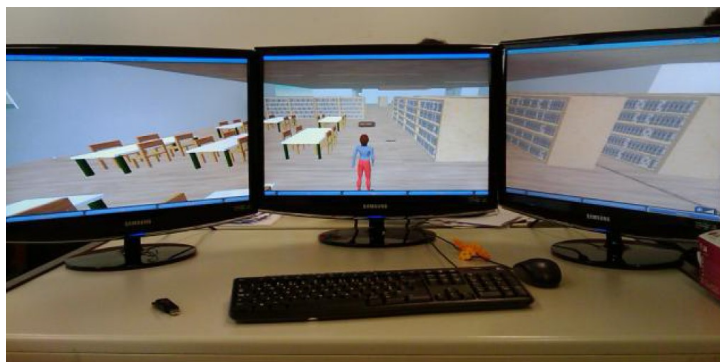

Figure 5.8: CaveSL with 3 clients running at the same time (adapted from [82])

5.2 Virtual Environment Creation

In the previous sections a number of viewers supporting different styles of interaction with the virtual world were described. However a fundamental prerequisite for using these solutions is to develop a virtual environment. This section presents this creation process.

The process of developing a rapid prototype in APEX includes the creation of a virtual environment close enough to the physical target system providing an adequate and realistic experience for users. An environment is created for the user or users by means of a viewer through the OpenSimulator server. The smart library example was created without using *mesh* objects. All the objects of this example were developed within the viewer. Creating the environment first involved developing a flat terrain where the building containing the ubiquitous environment was to be situated. The process of creating sophisticated elements such as chairs, bookshelves and tables was accomplished by linking basic shapes. Figure 5.9 shows the linkage of basic shapes, leading to the creation of a chair through the viewer. At the end of the creation process a virtual environment is created. Figure 5.10 shows snapshots of the use of the library prototype.

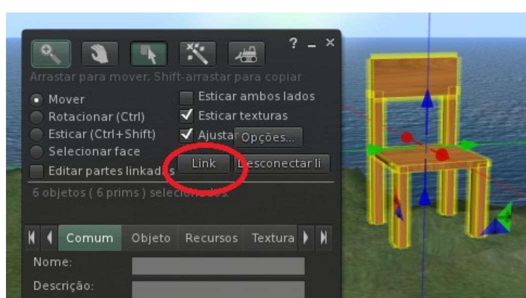

Figure 5.9: Linkage of the elements composing a chair

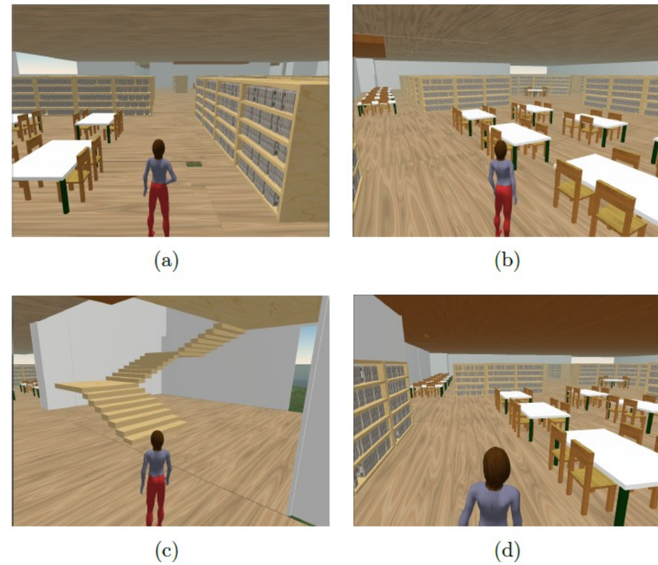

Figure 5.10: Smart library prototype created in the OpenSimulator server

Alternatively, it is also possible to create a simulation using objects that are already available. This can be accomplished by uploading them into the environment. Figure 5.11 shows the upload of a *mesh* object using the Project Mesh viewer. Once uploaded the object can be moved to the desired location. This is done using viewer features provided for object manipulation (e.g. object movement, scale and rotation). Once the virtual environment is created the space can be explored by simulated users using avatars.

5.3 Conclusions

The aim of this chapter was to explore alternatives to improving user experience while interacting with the prototyped environment. Solutions to improve different aspects of the experience include, stereoscopic 3D and multi-display support. Although not addressed in this chapter (already discussed in Chapter 3.1), the use of external physical devices also helps immersion by providing a more natural style of interaction. By using these solutions we aim to provide a richer experience for users when experiencing the prototyped ubicomp environment. How much immersion is enough and when simpler solutions provides the same user experience results has been discussed by Bowman & McMahan [84].

It is important to evaluate environments in terms of aspects that are difficult to address with exhaustive model analysis (e.g. *experience*). For this purpose it is of major importance

to use a virtual environment representation of the system under development. The support for exploring these spaces in different ways aims to provide users with a more complete experience, as close as possible to the one provided by the target physical ubicomp environment.

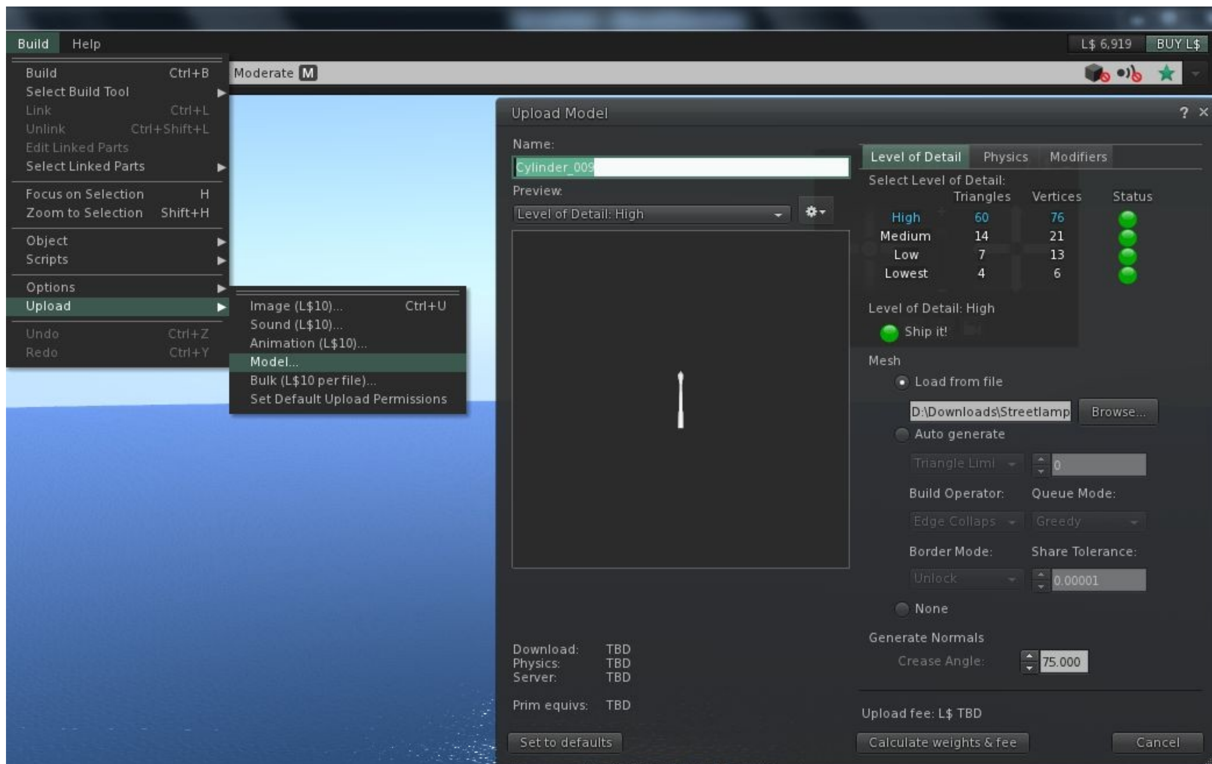

Figure 5.11: Mesh upload through the Project Mesh viewer

Chapter 6

Ubiquitous Environments Analysis

Ubicomp environments are difficult to analyse because of the large number of different elements involved in their construction, including features of ambient intelligence and distributed computing. As a result of the complex interactions arising from the combination of multiple sensors, devices and users in a physical space, observation of episodic use of the prototype alone is not sufficient to guarantee that some particular feature of the system is a general property of the design. It becomes difficult to establish whether observations made about use are truly representative of all possible interactions or whether certain characteristics of the system are true in all possible scenarios.

In APEX the behaviour of the ubiquitous system simulated within virtual environment is described using a CPN model. The fact that behaviour is driven by a CPN model makes it possible to analyse the behaviour of the prototype systematically and exhaustively using CPN Tools. The model-based approach makes it possible to reason about systems providing an adequate base for analysis. It is this analysis of ubicomp environments that forms the discussion of this chapter.

Because of the multilayer approach (Section 3.1.2), APEX prototypes can be analysed with real physical sensor data. The results of experiments, using the virtual and physical layers, help to focus analysis to particular contexts. These contexts relate to conditions that are likely to be encountered at runtime. Tools such as Replay [85] can be used to capture and play back sensor traces reflecting realistic scenarios that can be either analysed or tested within the modelling layer or alternatively experienced or tested at the simulation layer. Hence the design can be explored interchangeably at three different layers (modelling, simulation and physical). Common analysis techniques as well as the use of empirical and hypothetical data are easily usable as will be demonstrated.

Despite a possible behaviour being apparently correct at the simulation layer (by user exploration through the 3D simulation) it is useful to evaluate important aspects exhaustively thereby avoiding erroneous deductions. These situations can occur because, in experiencing the environment, the analysis is not exhaustive and consequently some features can work for specific uses but fail for different ones. Analysis, accomplished at the modelling layer, provides an accurate and exhaustive examination from which errors/faults are highlighted with little effort. For example, when experiencing the prototype at the simulation layer, the gate can behave correctly when an avatar gets close to it. However, the behaviour of the gate can fail when several avatars get close to it at the same time. At the simulation layer several avatars are needed to test this situation. In the modelling layer this situation is analysed just by adding values (e.g. user identifiers, presence sensors detections) to the scenario that is used to represent avatars close to the gate.

The application of special purpose heuristics to the design of the ubiquitous environment is the basis of the discussion. The description of the proposed approach starting from developed modules through the process that leads to property verification is presented. Finally, alternative analysis approaches are suggested.

6.1 Approach

The heterogeneous features of ubicomp environments pose challenges to developers in analysing them. As stated the model-based approach makes systematic analysis of the environment behaviour easier because technical aspects are abstracted. Being able to guarantee and check properties are the focus of concern here. There are other dimensions of analysis (e.g. user experience) that are not directly covered here.

Usability heuristics [68] are a starting point for analysis using the APEX system. In this approach the analyst is encouraged to explore how well a particular design supports general properties that encourage ease of use. The analyst or team of analysts bring their expertise in human factors or their understanding of the domain to decide where there are issues (for example in relation to ease of recovery, or the visibility of the effect of explicit or implicit actions) in the design that deserve being investigated through verification. When the verification detects problems, they can then propose design improvements.

6.1.1 Tool support

To achieve systematic and exhaustive analysis of the CPN behaviour model, the verification capabilities of CPN Tools are used. These tools provide a modelling and verification environment for Coloured Petri Nets. Particularly relevant here is the State Space (SS) tool. Reachability graphs (also known as state spaces or occurrence graphs) and a number of implemented standard queries are the means provided by this tool to accomplish the analysis. The tool generates a *reachability* graph that defines the states that can be reached from some starting state. Each node of the graph represents an execution state. Each binding of the variables associated to a transition is represented by an arc (the notion of binding is explained in Section 2.2.2). Figure 6.1 illustrates part of one of these graphs. The whole graph represents all possible executions of a ubicomp system showing which actions can be executed in each system state. Nodes and arcs are numbered and their values can be consulted (e.g. variable *idRead="removed"* of the arc caption in Figure 6.1). Arc captions indicate the selected binding which triggered the transition. Node captions indicate the tokens present in all places of the model for that particular execution state (thus describing the system state).

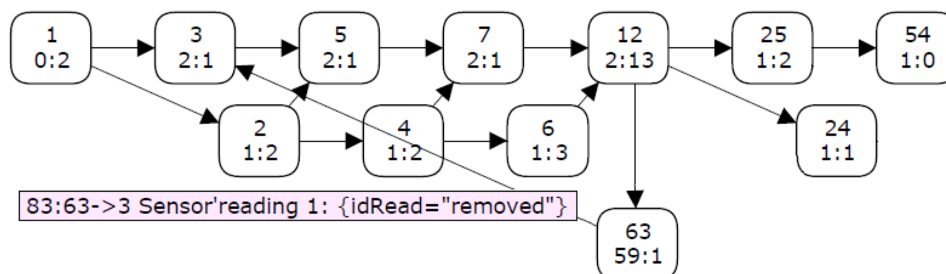

Figure 6.1: Reachability graph

The process of verification of a property involves applying a query to relevant states in the reachability graph. These queries are used to explore standard properties like reachability, liveness, fairness and boundedness [58]. Queries are used to request information about the generated reachability graph that demonstrate the truth of corresponding properties. Standard queries together with ML code are used to write specific queries about the CPN models, for example to demonstrate that the system always works as expected. The returned result is either that the query is true of all relevant states or that the query fails to be true, in which case the path to the failing state is deduced. This path can then be used to explore situations that

may be of interest from the perspective of the design of the ubiquitous system. More details about these tools are contained in the CPN Tools State Space manual [44] .

In APEX the CPN model (composed of modules) represents the behaviour of a ubiquitous environment and is the basis for exhaustive analysis. The SS tool creates a reachability graph computed up to some limit (e.g. number of nodes/arcs, time), executing a deterministic CPN model with all possible bindings. To create the graph, the *colour sets* defining the values used in the bindings should be finite. During the graph creation process a node is created whenever a binding leads to a new state of the model. Otherwise the binding is represented as an arc connected to the existing node that represents the reached state of the model.

6.1.2 Patterns

The approach uses verification patterns adapted from standard properties based on both usability heuristics and a broader range of properties used in other fields [86]. A particular set of property patterns is provided by the IVY tool [87]. This tool is a model-based usability environment for the analysis of interactive systems. In IVY, patterns define *property templates*, expressed in temporal logic, which must be instantiated to the particular details of the system and property under consideration. This instantiation process creates a temporal formula that can be verified. Analysis is performed automatically on the developed models using a model checker.

Applying the patterns in the context of APEX raises a number of challenges. A first challenge is how the property template, defined within the pattern, relates to the verification process. As explained above verification is achieved using the SS tool by writing queries over the reachability graph. Hence, the pattern, instead of defining a temporal logic template, must define an algorithm skeleton describing how the reachability graph is to be explored (which queries are needed) to perform the verification. Other challenges concern the interpretation of the patterns, and in particular:

- Who are the users? IVY patterns assume interaction between a user and the device. In APEX the interaction context is richer, involving spaces where several *users* might be present. Hence, when considering user actions and system responses it is necessary to consider how different users affect each other, e.g., an action by one user might trig-

ger a system response directed to a different user. It becomes relevant to consider therefore who carried out an action or caused some change in the system state.

- What are the actions? In a ubicomp setting implicit interaction becomes relevant as well as explicit user action. The system might be responding to conditions arising through implicit user action or environmental change. These conditions are typically monitored indirectly through sensors, e.g., a user entering or leaving a room. Hence, rather than actions, *situations* of interest may require characterisation.
- What is being analysed? A general problem not specific to this context is whether the property is addressing the design of the system or the model itself, i.e., whether the property is being used to reason about features of the system's design, or is being used to validate the model itself. This affects the interpretation of the reachability graph. Indeed, while some nodes correspond to states of the ubiquitous system, others correspond to intermediate execution states of the model.

In this chapter and the next these interpretation challenges are clarified with concrete examples.

6.2 Setting Up the Analysis

APEX performs formal analysis of models using the SS tool. To be able to accomplish analysis with APEX some steps are necessary. This section shows them. The model conversion, the scenario selection for analysis using the purpose built tool APEXi and finally the usage of the SS tool allows verification of properties based on the generated reachability graph.

6.2.1 Model conversion

For a property to be verified, the model must be converted to a form that will allow CPN Tools to check the truth of the property. CPN Tools require that the model be deterministic and small so as to reduce the search space used during analysis. The SS tool then uses brute force to bind each variable to each of its possible known values, creating the reachability graph.

A model is considered to be nondeterministic, as defined in CPN, when the execution of an enabled transition can lead to a state that cannot be uniquely determined by the binding of the transition's variables. The use of random distribution functions (e.g. *Poisson* and *Student*) or the reading of unpredictable values (external information exchange) leads to these situations. In both situations the bindings of the transition's variables cannot be uniquely determined and are therefore unpredictable. The application of the SS tool to a nondeterministic model is not feasible because it is not possible to create a constant reachability graph in different SS tool executions (states cannot be identified deterministically). A model conversion must be accomplished prior to SS tool usage. This conversion removes non-determinism by substituting two types of functions (distribution functions and external information exchange functions) by reading from a finite set of values (a *small colour set*). CPN Tools define a set of up to one hundred elements to be a *small colour set*. In the context of distribution functions, if there is a known limited set of all possible values that are used or are significant these random distribution functions can be substituted by *small colour sets*. Consequently, the SS tool can use brute force to consider all different possibilities. The same happens with information exchange functions that are substituted by functions that read from *small colour sets* composed of the possible values to be read.

Because, under normal conditions, an APEX model exchanges information with the virtual world simulation and/or actual physical external devices, the model is nondeterministic and can in principle be of unlimited size. This large open model must therefore be translated into a closed one, to make it tractable within the SS tool. Closing the model means isolating it from external components. As explained above, this is achieved by defining finite sets of possible values for all the variables in the model that previously held values acquired externally. A small example is presented to illustrate this conversion process.

Figure 6.2 and Figure 6.3 illustrate the conversion of a non-deterministic module to a deterministic one. This module is responsible for reading information sent from the other layers and to analyse it to decide which module of the model will process it. The *receiveString* function is used to read external data (Figure 6.2). The function is substituted by the reading of values from a *small colour set* (Figure 6.3) composed of 13 different values (the values present in the box) making the module deterministic. This is accomplished by introducing a place (*Commands*) initialized with a list of all possible values to be read. This list is used by the transition and values to be processed are selected randomly (e.g. *newUser*) and bound to

the *id*. The *run* place (see Figure 6.2) was removed because it is now the case that before the reading of values from the *small colour set* these have to be initialized. The *run* place was added to the *small colour set* initialization modules that are first executed.

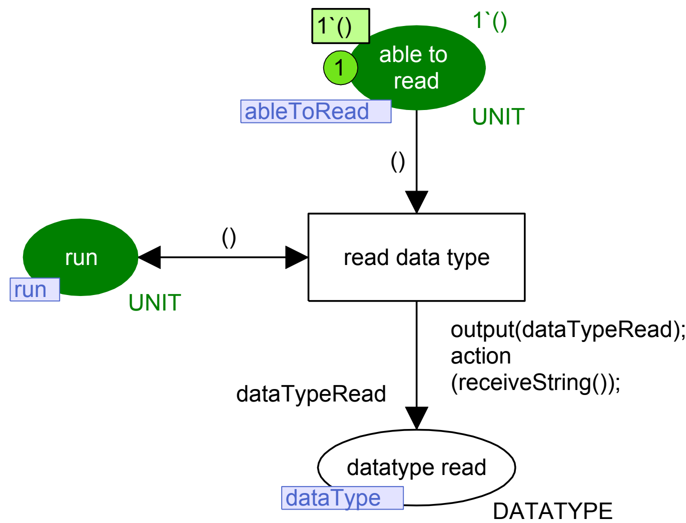

Figure 6.2: *DataTypeReading* non-deterministic module

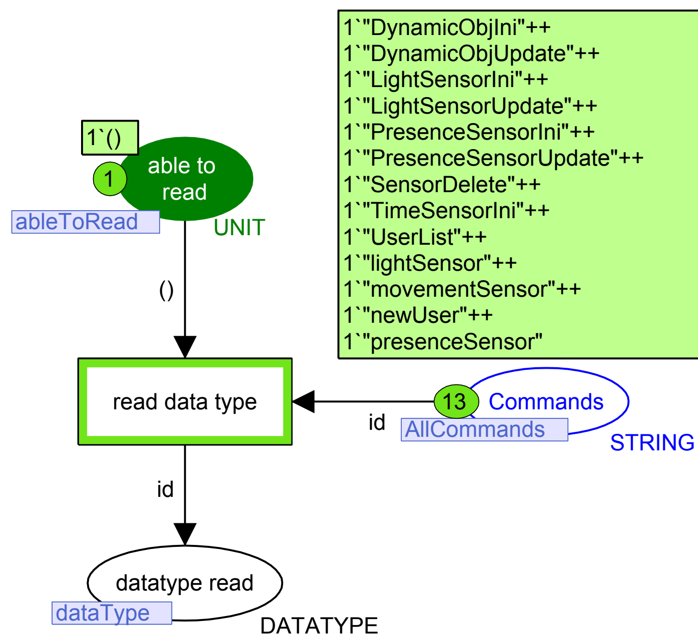

Figure 6.3: *DataTypeReading* deterministic module

The problem of this conversion appears when the set of possible values exchanged with the other layers is larger than 100 (upper limit size of *small colour sets*). For instance the set of values used by *timeSensor* (senses the actual time) and *movementSensor* (senses user movements) typically consist of more than one hundred values. This means that after having proceeded with the model conversion, selection criteria must be taken into consideration to choose which values will be present in the *small colour sets*. For example, it might only be relevant to consider when the user arrives at or leaves the area covered by the movement sensor. In this case movements while not near the sensor do not need to be considered

6.2.2 APEXi tool - scenario selection and *small colour sets* initialization

APEXi, a component of APEX, is used to initialize small colour sets semi-automatically. The tool provides an interface to enable analysts to supply or select desired values to populate with tokens the corresponding places of the CPN model as represented by a chosen scenario. Different scenarios can be tested by passing different values to the model as input using the APEXi tool. This tool and its usage is described in this section.

Analysis Models

The amount of work associated with model conversion makes the analysis process relatively slow. For this reason the conversion process was simplified. The simplification introduced is that only specific modules developed for a particular situation need to be converted (e.g. gate module, screen module), for all other modules the corresponding closed versions are provided. Two CPN base models for the use of developers are provided as part of APEX. The original *CPN base model* (Appendix A) drives the virtual environment and is used for user exploration of a proposed design. A closed version of the *CPN base model* the *CPN analysis base model* (Appendix B) is used for analysis. This analysis model is a semi-closed version of the original one. It uses *small colour sets* instead of reading functions and contains modules responsible for *small colour sets* initialization. However, these modules include some reading functions to read values from APEXi used to fill the *small colour sets*. Despite having these reading functions and being potentially non-deterministic, this semi-closed model is still adequate for analysis because after the *small colour sets* are initialized the reading functions pre-

sent are no longer executed. Consequently, this model is equivalent to providing a deterministic model to the SS tool with the advantage that it has the capability of initializing the *small colour sets* present. Notice that this model is usable for analysis with the SS tool only after the initialization of the *small colour sets*.

Architecture and User Interface

The second step after model conversion consists in scenario selection. The APEXi tool was developed to reduce configuration effort prior to analysis and to satisfy the need for CPN *small colour sets* initialization. The selection of desired values representing a scenario is established semi-automatically using this tool. APEXi enables selected and provided values to be inserted automatically into the respective *small colour sets* of the deterministic model making it ready to be tested. Useful situations, i.e. the ones that occur in practice through typical use can easily be recreated and tested avoiding typical verification problems associated with considering all possible values (e.g. state explosion). The selection of adequate values for analysis is an important step that the analyst must consider carefully.

The APEXi tool aims to reduce the time needed to create scenarios to be tested as well as increasing analysis automation. It reuses part of the APEX *communication/execution* component responsible for exchanging information with CPN models. Modules that receive values sent by the APEXi tool also use functions of the Comms/CPN library [77] that connects CPN Tools with external processes as done in APEX (see Section 3.1.1). The Figure 6.4 illustrates how APEXi tool relates to the remaining components through connection to the APEX *behavioural component*.

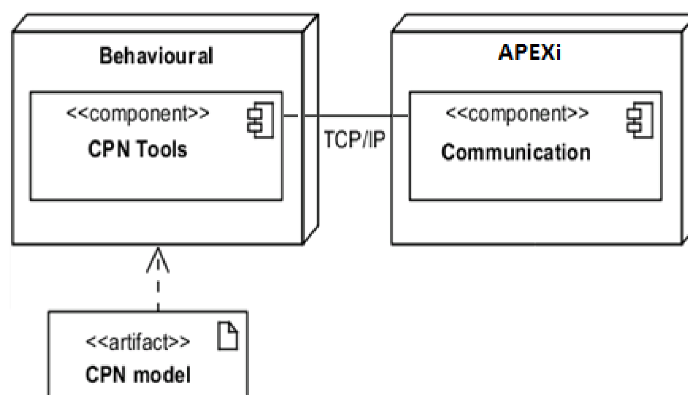

Figure 6.4: APEXi tool connection to the APEX *behavioural component*

APEXi supports *small colour sets* initialization and its graphical user interface (GUI) is presented in Figure 6.5. The GUI provides support for the enrichment of the *small colour sets* by providing *lists* and *checkboxes* that enable the selection of the scenario to be considered for analysis using the model. For example, the *UsersIDs* list (Figure 6.5) enables the insertion of the users' names (e.g. *Silva* and *Cardoso*). The interface contains a collection of *lists* that are labelled with the names of all *small colour sets* that can be populated with the values provided. The *small colour sets* are always the same considering the type of elements provided (e.g. presence sensor). Depending on the particular scenario being analysed some *small colour sets* do not need to be initialized (e.g. presence sensors not used) or additional fields used to initialize the *small colour sets* associated with the new type of elements may be required (e.g. use of new type of sensors). The association between the lists (e.g. which users a presence sensor affects) is specified sequentially i.e., the position of the values in the lists (line number) is considered and used to make associations.

To illustrate APEXi a scenario involving two users is analysed (see Figure 6.5). The values that represent their behaviour in the scenario are inserted in the APEXi interface so that they can form the relevant *small colour set*. Some elements of the *InitialCommands* field are checked. This means that, in this scenario, the connection of the users to the simulation (*newUser*), the initialization by the analyst of the presence and light sensors (*PresenceSensorIni*, *LightSensorIni*) and the synchronization of the dynamic objects with the value provided (*DynamicObjIni*) are simulated.

In the list *AllCommands* the *presenceSensor* field is checked enabling the simulation of the detection of users by the presence sensors. The *lightSensor* is checked enabling the simulation of the light level variation of the environment. The values of the light are associated with the hour of the day provided in the *Hours* field. In this example three levels are possible (1, 2 and 3). The *UserList* is checked enabling the variation of the elements that compose the user's list.

The *ObjIDs* field makes it possible to specify the identifiers (must be numbers) of the dynamic objects present. Their features are specified in the *OBJfeatures* lists. In this scenario the object present is in the position (x=121, y=122, z=120) and it is a window (*Type* field value). Two users are present (*UserIDs* field). The user *Silva* has an associated list with varying content (*Listfeature* lists). The value 1 (*Values* field) can be added or removed (*PUT* and *REM* values of the *Actions* field). To associate values to the list of the *Cardoso* user the sec-

ond line of the *Listfeatures* lists should be completed. In this scenario two presence sensor are present with identifiers *PS1* and *PS2* and with the actions to detect new users (*PUT*) and clear the list of detected users (*CLEAR*). This information is present in the *presenceSensorsIDAction* field. The list of users to which these sensors react are present in the *S_UserIDs* field. The information is associated sequentially. Consequently the presence sensor *PS1* reacts to the user *Silva* and the presence sensor *PS2* reacts to all users present (*ALL value*).

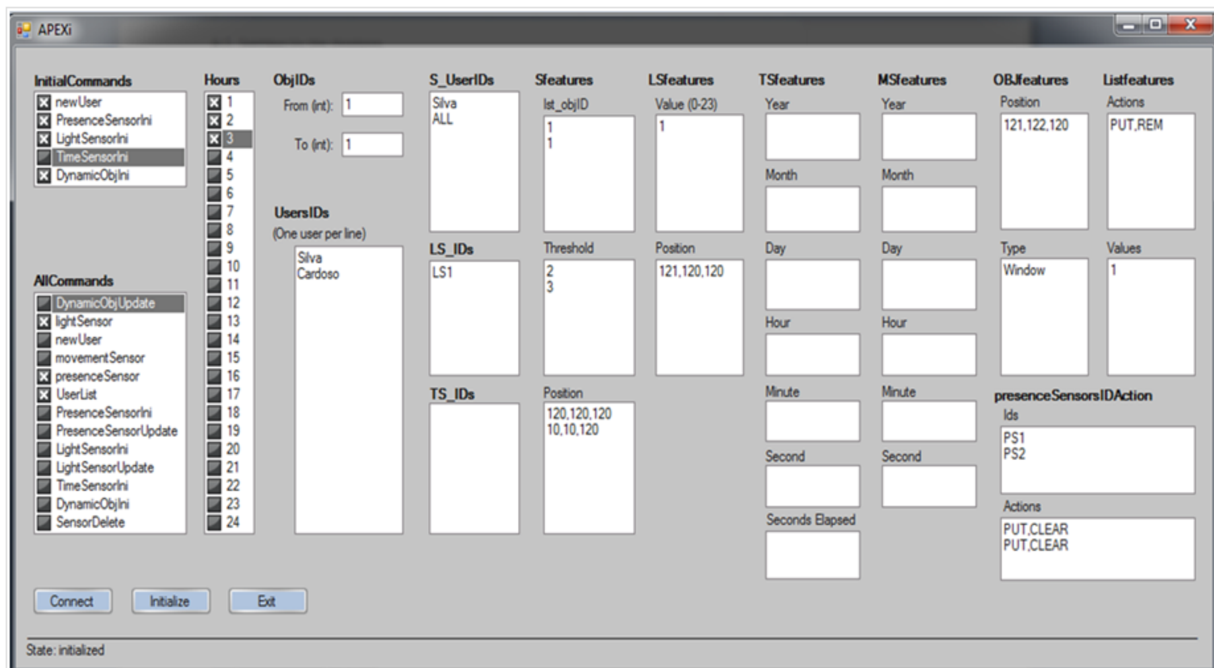

Figure 6.5: APEXi interface

This scenario is also composed by one light sensor (*LS1*). One identifier is specified in each line. The *Sfeatures* lists are used to specify the features of each presence sensor. In this case the presence sensor *PS1* affects the object with id one (first line of the *lst_objID* field), has a threshold of 2 (first line of the *Threshold* field) and is in the position x=120, y=120, z=120 (first line of the *Position* field). The presence sensor *PS2* affect the object with id one (second line of the *lst_objID* field), has a threshold of 3 (second line of the *Threshold* field) and is in the position x=10, y=10, z=120 (second line of the *Position* field). Analogously the list *LSfeatures* is used to specify the features of the light sensors. In this case the light sensors has initially the value of one (*Value* field) and the position x=121, y=120, z=120 (*Position* field). Analogously the lists *TSfeatures* and *MSfeatures* are used to specify the features of the

time and movement sensors. Since these sensors are not used in this scenario these lists are empty.

The values selected and provided specify the sequence of actions that happen in the scenario. A description of each element of the APEXi GUI is presented below. The APEXi GUI has the following elements:

1. *InitialCommands*: offers the list of commands to be executed before starting the exploration. The *newUser* option simulates the connection of a new user to the simulation and is the only command of the list (see Figure 6.5) which is not an APEX command provided to users in the exploration of the environment through a viewer, see Appendix E for a list of commands provided by APEX to users. As explained in Appendix E these commands are used to synchronize components of the simulation with the model (e.g. sensors) before the exploration. For example it does not make sense to execute the *PresenceSensorUpdate* command without executing the *PresenceSensorIni* command first;
2. *AllCommands*: offers the list of all commands to be executed continuously throughout the exploration once the initial commands have been executed. Some commands are provided by APEX (e.g. *PresenceSensorIni*, see the Appendix E) while others represent events in the simulation (e.g. *PresenceSensor*). The users' positions are not represented in the model. Rather, identifiers of users detected by presence sensors are used. Consequently, the use of the *PresenceSensor* command simulates the reading of new user identifiers, which represent the movement of avatars in the scenario. The list of detectable users is provided in the *S_UserIDs* list (see description below);
3. *Hours*: offers the list of possible hours being used during the exploration enabling the consideration of time in the specification of a scenario;
4. *ObjIDs*: makes it possible to specify the list of dynamic object identifiers present. It makes it possible to indicate which dynamic objects compose the scenario;
5. *UsersIDs*: makes it possible to specify the user identifiers present;
6. S_*UsersIDs*: user identifiers that presence sensors react to. Used to specify to which users a presence sensor reacts to. Each line in the list corresponds to the presence sensor in the same line in the presence sensors' IDs list (first list of the *presenceSensorsIDAction* lists);
7. *LS_IDs*: identifiers of the light sensors that are present;

8. *TS_IDs*: identifiers of the time sensors that are present;
9. *Sfeatures*: a set of three list boxes (*List_objIDs*, *Threshold* and *Position*) describing the features of each presence sensor. The features of each sensor are represented in its corresponding line (the first line of each list box corresponds to one sensor, the second line of each list box corresponds to a different sensor, etc.). These features are the following:
    a. *List_objIDs*: list of object identifiers which the presence sensor affects;
    b. *Threshold*: distance from which the presence sensor triggers an action;
    c. *Position*: the position of the presence sensor (as stated, each line refers to the features of the corresponding presence sensor).
10. *LSfeatures*: describes the features of each light sensor. Their features are the following:
    a. *Value*: list of light sensors values from 0 to 23 (representing the light level throughout the day). These are used to modify the intensity of the light of the scenario;
    b. *Position*: positions of the light sensors.
11. *TSfeatures*: list of time values (year, month, day, hour, minute, second and elapsed seconds) of the time sensors. For example, dates are used in the model to associate them to an action. In the figure only one date is present but more dates can be added (one date per line).
12. *MSfeatures*: list of time values triggered by the movement sensors. Used to identify the time of specific movement in the scenario;
13. *OBJfeatures*: features of the dynamic objects present.
    a. *Position*: positions of the dynamic objects;
    b. *Type*: type of dynamic objects.
14. *Listfeatures*: values associated with the users (users' list).
    a. *Actions*: possible actions to perform in the users' list, add an element (*PUT*) or remove (*REM*);
    b. *Values*: list of information to be associated with each user through the actions. In the figure the number 1 indicates the identifier of the book (present in the *ObjIDs* field) desired by the user *Silva*. Each line in the list corresponds to values of the user in the corresponding line (field *UsersIDs*).

15. *presenceSensorsIDAction*: describes information associated with the presence sensors.
    a. *Ids*: list of presence sensor identifiers;
    b. *Actions*: list of presence sensor actions. The action *PUT* adds user identifiers to the list of users detected by the sensor. The list of detectable users is provided in the presented *S_UserIDs* list. The action *CLEAR* removes all the elements from the identifiers detected list and the action *REM* removes one element. As stated this is used to simulate user movement.

The interface also contains 3 buttons to:

i) establish the connection between APEXi and the model (*Connect*);
ii) start the values initialization process (*Initialize*);
iii) leave the application (*Exit*).

The Initialization Process

A more detailed description of the initialization process is now presented. The connection request between APEXi and the CPN model is started by pressing the *connect* button in the APEXi interface. The connection is accepted at the modelling layer by the module of Figure 6.6 which forwards received values to the *dataTypeAPEXi* fusion place. After the communication has been established the populating process can start. This process is not straightforward. Figure 6.7 shows a *small colour set* initialization module. This module reads the type of values transmitted from APEXi present in the *dataTypeAPEXi fusion* place. These values are sent one by one. Depending on the tokens held in the place different transitions may fire. There are different modules to initialize different *small colour sets*. Figure 6.7 represents a module that initializes the object and user identifiers. It initializes two *small colour sets*, *ObjIDs* and *UsersIDs* and it is executed when values of these types are sent by APEXi. All types of elements present in the APEXi interface are supported. APEXi sends each selected value to the CPN model preceded by its type (e.g. *UsersID Silva*). In the example if the type of the value sent is equal to *UsersID*, then a token representing the user identifier sent is inserted into the place which holds the *small colour set* of all user identifiers (place *UsersIDs*). The value of the inserted token is obtained by the returning value of the *receiveString* reading function associated with the transition. After that, a token is inserted into the *ready2ReadAPEXi fusion* place enabling the populating of other *small colour sets* (Figure

6.7). The initialization process stops once all values are forwarded to their corresponding *small colour set*.

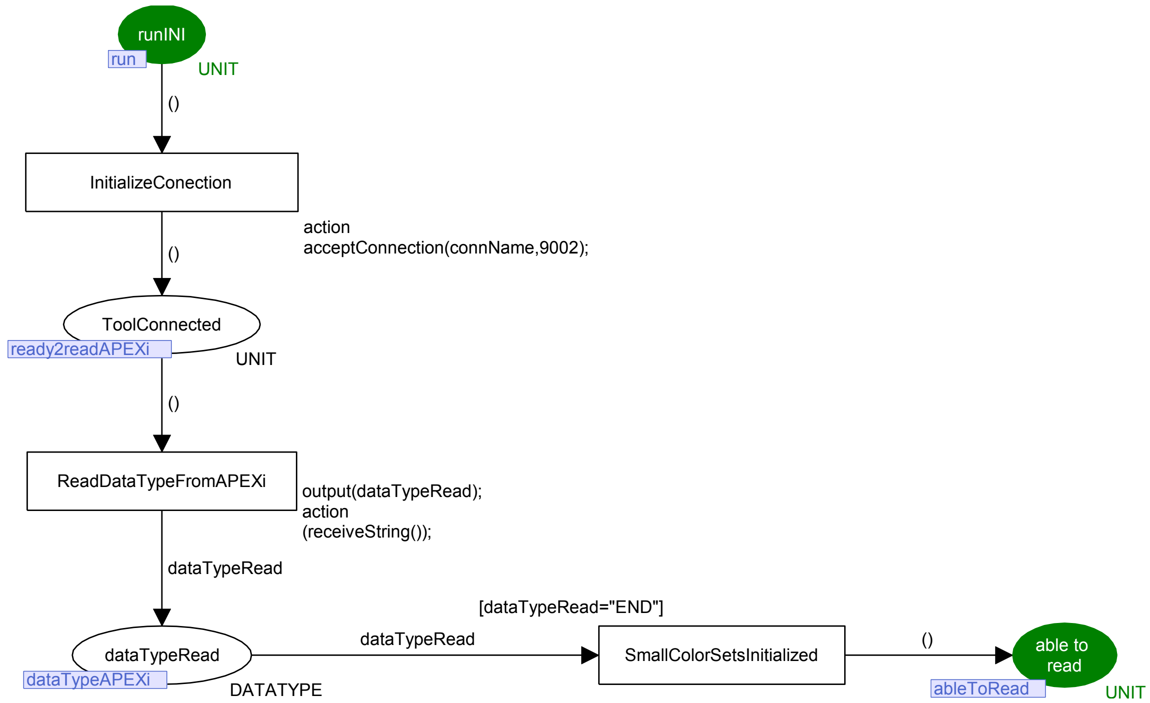

Figure 6.6: APEXi and CPN model connection module

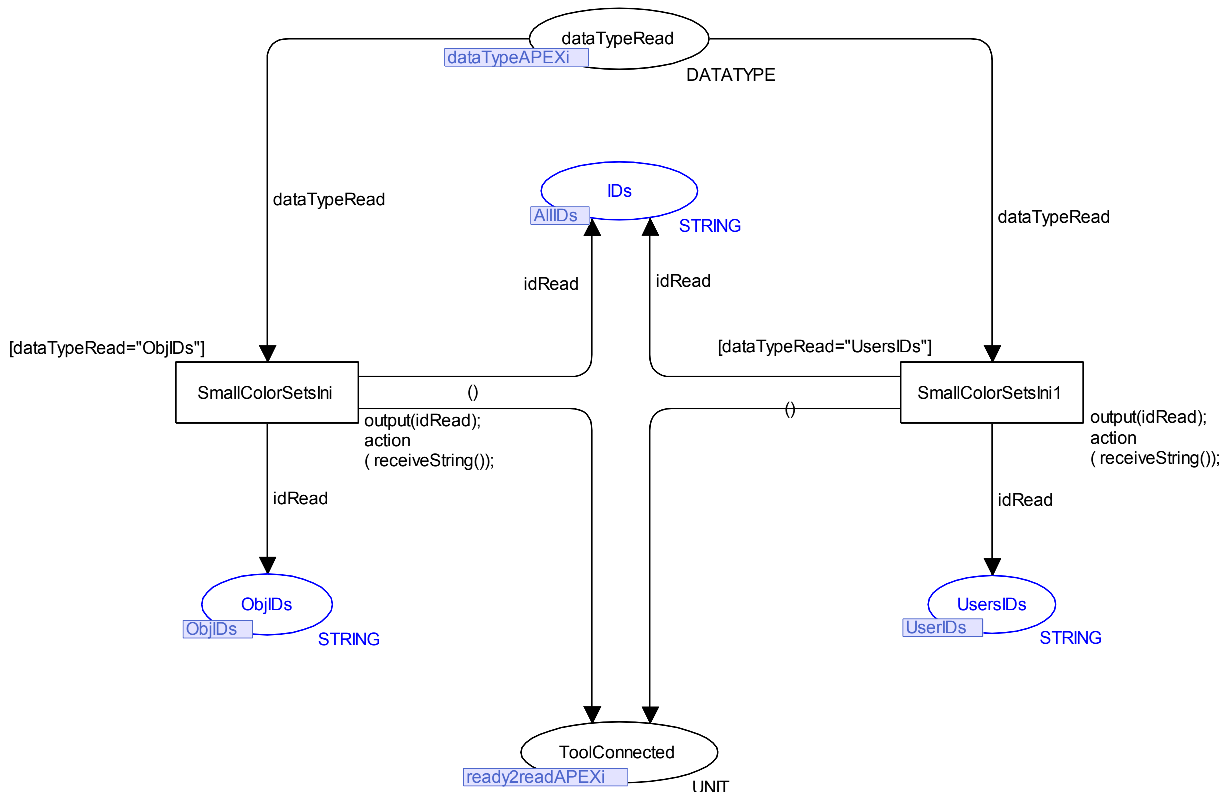

Figure 6.7: Module which initialize some *small colour sets* (e.g. *UsersIDs*, *ObjIDs*)

The selected values simulate the information that can be read from the simulation and physical layers for a particular situation. Providing these values makes it possible to create specific scenarios and to analyse them avoiding the need for a time consuming exhaustive analysis that arises as a result of the state explosion that results from the large set of values.

The prototype developer is the one who transforms a scenario into values of the APEXi tool to be analysed. The different features supported by APEXi have been presented. An illustration of how a complete concrete example is specified in APEXi is presented in the next chapter.

Once all *small colour sets* are initialized the model can be analysed exhaustively with the SS tool. The next section presents the instruments provided by the SS tool for this exhaustive analysis.

6.2.3 Reachability graph and analysis

This section describes first a resulting reachability graph and how it is interpreted. Secondly the concepts concerned with the SS tool for analysis are presented through a simple example.

Reachability graph

Applying the SS tool to the initialized model creates a directed graph with all possible executions of the model as limited by the *small colour sets*. Figure 6.8 provides a partial representation of one of these graphs. The selected values for the initial commands in the example were *newUser*, *presenceSensorIni*, *TimeSensorIni* and *DynamicObjIni* as shown in Figure 6.9. Consequently node 6 of Figure 6.8 has four successor nodes. These nodes represent each of the possible bindings. As an illustration, arc number 9 (from node 6 to node 10) represents the binding of the variable *id* (which is used to identify the command read) with the value *DynamicObjIni*. This information is represented in the arc caption (*{id="DynamicObjIni"}*) in Figure 6.8.

As stated the value of each node and arc can be consulted and by this means it is possible to identify the values of a state (the tokens present in each place of the model), in the case of nodes, or the binding value of variables, in the case of arcs. Figure 6.10 shows the exploration of node 10 in the reachability graph. The second line of the figure indicates that there is one

token in the *datatype_read* place as expected (*1`"DynamicObjIni"*). It is also possible to know that the place *UsersIDs* of the *MovementSensorUpdate* module contains two tokens, *Other* and *Test User*. This is deduced by consulting the sixth line of the Figure 6.10 (*MovementSensorUpdate'UsersIDs 1: 1`"Other"++1`"Test User"*).

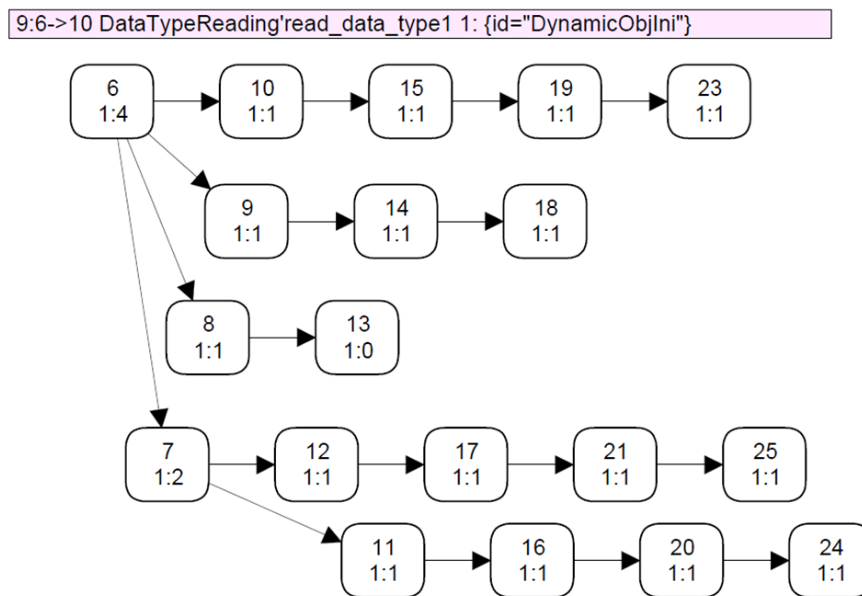

Figure 6.8: Reachability graph

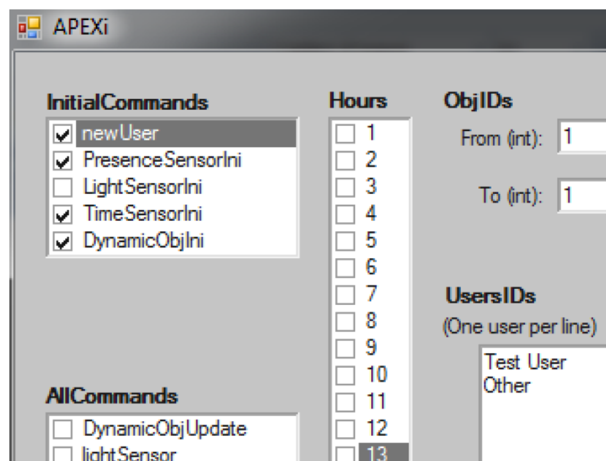

Figure 6.9: APEXi selected values

```
10:
MovementSensorUpdate'datatype_read 1: 1`"DynamicObjIni"
MovementSensorUpdate'read_ids 1: empty
MovementSensorUpdate'able_to_read 1: empty
MovementSensorUpdate'M_sensor 1: empty
MovementSensorUpdate'UsersIDs 1: 1`"Other"++
1`"Test User"
MovementSensorUpdate'MS_features 1: 1`{year=[2010],month=[2],day=[22],h=[4],m=[10],s=[7]}
InitializeLS_IDsTS_IDs'dataTypeRead 1: empty
InitializeLS_IDsTS_IDs'ToolConnected 1: empty
InitializeLS_IDsTS_IDs'TS_IDs 1: 1`"TS1"
InitializeLS_IDsTS_IDs'LS_IDs 1: 1`"2"
InitializeLSfeatures'dataTypeRead 1: empty
InitializeLSfeatures'ToolConnected 1: empty
InitializeLSfeatures'LS_features 1: 1`{h=[21],x=[120],y=[120],z=[120]}
Sensor'datatype_read 1: 1`"DynamicObjIni"
Sensor'read_ids 1: empty
```

Figure 6.10: Reachability graph node consultation

Properties - APEXi and SS tools

Properties are checked by means of the reachability graph. They are expressed by identifying states over the reachability graph that are relevant to determining whether the system behaves as expected (e.g. reachability property). Consider a property of a simple ubiquitous system that helps disabled elderly people to go to the bathroom at night by showing the way using lights that are lit as determined by sensors. The lights come on when the user is near a presence sensor and go off after a certain delay, when the user is near to another presence sensor. Developers may wish to prove that the light always goes off when the person is near another presence sensor (consistency property). A way to prove this could be by defining a query that searches for states where this property does not occur. If no states are found then the property is proved. If states are found, the property is false, and those states represent counter examples (situations where the user is near to another presence sensor and the light remains on). Basic properties are demonstrated in this section in order to introduce the SS tool capabilities. Chapter 7 verifies a set of potentially useful properties (e.g. consistency, feedback, precedence). The next section presents proposed property patterns for ubicomp environments to be followed in order to identify and check properties more easily.

In the example above only one module (responsible for modelling the lights' behaviour) needs to be added to the *analysis base model*. This module (presented in Figure 6.11) is composed of four *fusion* places (*objects, users, timeSensor* and *presenceSensor*) used to determine when to turn lights on or off. For example, a light is turned on when a (registered) per-

son is near a presence sensor that affects this light. The condition *[is(obj,"light") andalso objAfectedByPresenceSensor(ps,obj) andalso userNearPresenceSensor(ps,u)]* associated with the *turn light on* transition is responsible for satisfying this condition. A similar condition is associated with the *turn light off* transition.

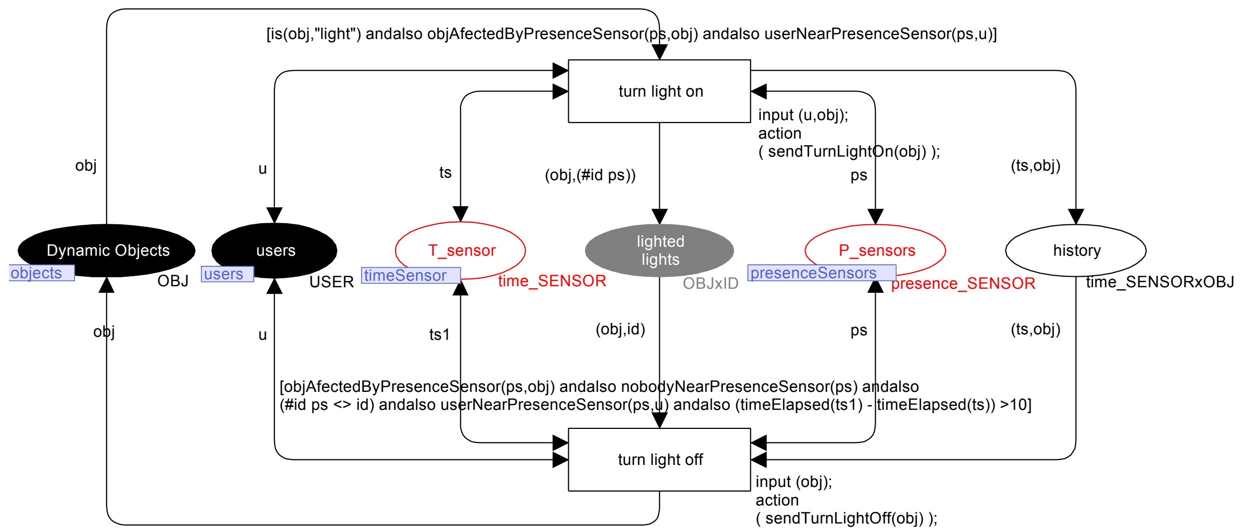

Figure 6.11: CPN light module

Consider now that the developer wants to verify the example property "*the module turns lights off when the user is near to another presence sensor*" for a specific scenario. A scenario must be provided to the APEXi tool to be analyzed by means of the SS tool. Then a query on the reachability graph is generated, based on the scenario and model, to demonstrate the property.

Initially a simple scenario with just one user (Figure 6.12a), one light (id=1, position x=121, y =122, z =123) (Figure 6.12b) and one presence sensor (Figure 6.12c) is used. The movement of the user is simulated by a series of presence sensor detections. For this purpose the *presenceSensor* command is used (Figure 6.12d). In the example, the date and time sensed are 2011-02-22 4:10:07 and 20 seconds elapsed (see Figure 6.12e). This information is used to determine when to turn off the light. The module specifies a minimum delay of 10 seconds to turn off a light. Consequently using this scenario the delay needed to turn off the light is satisfied because the provided seconds elapsed (20) are greater than 10.

After the initialization the next step toward verification is to specify a query that captures this property. The desired query is composed of two simple queries which, when executed,

enable the analysis of the stated property (if it is true or not). Breaking the property into two queries facilitates the verification. The first query identifies nodes that are then provided as arguments to the second query. These queries are responsible for the:

1. identification of nodes in which the light is on;
2. verification that the light is turned off from the nodes identified by the first query.

The first query identifies nodes of the reachability graph where the light is on (Figure 6.13). The query uses the predefined *PredAllNodes* query (described next) that looks for nodes that satisfy a provided condition. In this case the condition is that the number of tokens in the place (*lighted lights*, see Figure 6.13) that describe the lights as turned on is greater than zero. The variable *lightID* (Figure 6.13) holds the values (i.e. *id*, *position*, *objType*) of the light selected for the analysis (provided to APEXi). The execution of this query over the reachability graph (*lightNodes lightID*) returns the list of nodes in which the light is on (Figure 6.14).

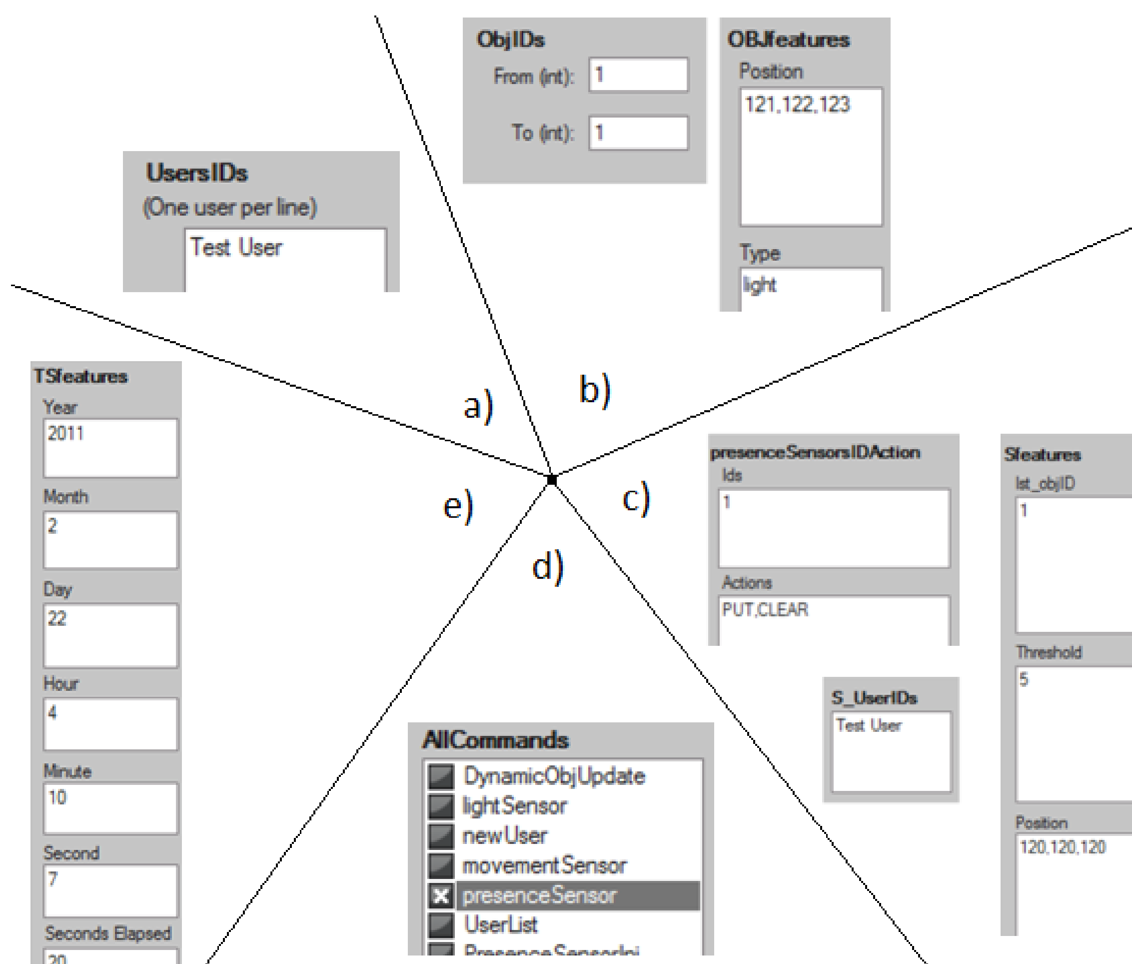

Figure 6.12: Specification of elements in the APEXi interface: a) one user; b) one light; c) one presence sensor; d) user movement simulation; e) one time sensor

```
val lightID = ({id="1",position={x=121,y=122,z=123},objType="light"},"1") : OBJxID

fun lightNodes(objID:OBJxID) : Node list
= PredAllNodes (fn n => cf(objID,Mark.ExplicitModel_light'lighted_lights 1 n) > 0)

val resL = lightNodes lightID
```

Figure 6.13: Lights' on query

```
val resL =
 [999,998,984,983,981,980,979,978,977,976,975,974,961,960,949,948,946,945,
  944,943,942,941,940,939,926,925,914,913,911,910,909,908,907,906,905,904,
  895,890,889,885,876,875,873,872,871,869,868,867,866,865,854,848,847,842,
  832,831,828,827,826,823,821,820,819,818,817,798,797,795,782,781,779,778,
  777,776,774,773,771,770,769,768,767,766,747,746,745,730,729,728,727,726,
  725,724,723,722,721,720,719,718,695,694,...] : Node list
```

Figure 6.14: List of nodes where the light is on

The second query identifies the nodes in which the light is turned off. The query looks for the states where the light is on (previously obtained, variable *resL*) and can be turned off, i.e. a light off state is reachable. The initial steps are to identify dark nodes (i.e. when the light is turned off). This is accomplished using the same reasoning as was used for the first query (Figure 6.15). Then the *Light2Dark* function identifies the nodes from which a *Dark* node can be reached. This function uses the predefined *SearchNodes* function (also described next) used to obtain desired states that satisfy specified conditions. The predefined *Reachable* function is also used to identify the reachability between two nodes.

```
fun darkNodes(objID:OBJxID) : Node list
= PredAllNodes (fn n => cf(objID,Mark.ExplicitModel_light'lighted_lights 1 n) < 1)

val resL = lightNodes lightID
val resD = darkNodes lightID

fun Light2Dark l =
 let
  val resL = lightNodes l
  val resD = darkNodes l
 in
   SearchNodes (
   resL,
   fn n => contains (map (fn x=> Reachable(n,x)) resD) [true],
   NoLimit,
   fn n => n,
   [],
   op ::)
 end

Light2Dark lightID
```

Figure 6.15: Light turns off property

The *Light2Dark* function execution over the graph returns an empty list. This means that for the developed module and selected scenario the light never turns off. The explanation for this situation is that the module developed only turns off a light when the user is near a presence sensor that was different from the one responsible for turning on the light. In the selected scenario only one presence sensor is used so it is obvious that the light will never turn off. As demonstrated the selection of adequate scenarios is essential for the adequate verification of properties.

With the introduction of a second presence sensor (Figure 6.16) the analysis produces a different set of results. A list of states is returned by the *Light2Dark* function where the light turns off in the new scenario.

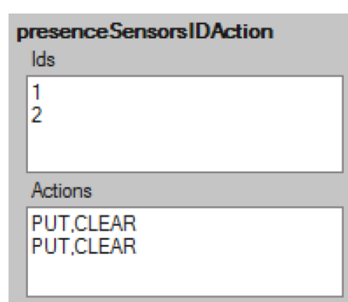

Figure 6.16: Inserting two presence sensors in the APEXi interface

Defining the queries is a detailed process that should not be done in an ad hoc way. An approach is needed to support query development. Property patterns have been developed to help in this development and are presented in the next section.

The standard *searchNodes* and *predAllNodes* queries used in the previous queries are now clarified. The *SearchNodes* query identifies nodes of the graph and takes six arguments used to filter and evaluate the nodes. These arguments are:

- *Search area*: indicates the part of the state space to be searched;
- *Predicate function*: used to ignore some nodes during the analysis;
- *Search limit*: indicates how many times a *predicate function* should evaluate to true before terminating the search;
- *Evaluation function*: application of a function to the nodes in the search area that satisfy the provided *predicate function*;
- *Start value*: initial value of the returned results, usually an empty list ([]);

- *Combination function*: a function that indicates how each individual result is combined with the collected ones, usually the concatenation function (::).

Some variations of this standard query are provided by CPN Tools with predefined arguments that are useful to express properties. The *PredAllNodes* standard query is one of them. This query is the instantiation of the *searchNodes* query with the following arguments:

- *Search area*: all nodes
- *Search limit*: no limit
- *Evaluation function*: identity function
- *Start value*: empty list
- *Combination function*: concatenation function

Reachable, *ListDeadMarkings*, *ListLiveTIs* are examples of other standard queries provided by the CPN Tools. The *reachable* query returns a possible path to get to the required destination state from the specified original one: "*After doing an action it is possible to return to the previous state*" is a concrete example of the use of the reachability query. The *ListDeadMarkings* query returns the list of dead nodes, specifically these nodes identify system states from which it is impossible to do anything. *ListLiveTIs* provides the list of live transitions present in the model.

In this section a simple example was used to introduce the concepts involved with analysis using the SS tool. More complex and useful examples are addressed in Chapter 7.

6.3 Property Specification Patterns for Ubicomp Environments

Patterns provide a basis for analysis serving two roles: they aid the process of elicitation of appropriate properties and they help the analyst use CPN Tools to perform the analysis of the instantiated property. A sample of relevant property patterns are now described. These patterns are taken from those supported by the IVY tool [87] applied to the ubicomp context (consistency, feedback, reachability, precedence, completeness, reversibility, possibility, universality and eventually patterns).

In [88] patterns are expressed as CTL (Computational Tree Logic) [89] templates to be instantiated to concrete actions and queries. We express the patterns as queries over the CPN model's reachability graph where actions are represented as transitions and effects as queries

over states. States are defined by the values of the attributes in the model and capture relevant configurations/conditions of the system.

All relevant patterns are presented following a common structure. Firstly the justification and intuition for the pattern is presented. Secondly its meaning in the ubicomp context is described. Finally a description of the algorithm skeleton to be used over the reachability graph to prove the property is provided.

6.3.1 The consistency pattern

Justification: Consistency is a heuristic that has widespread relevance, including in the ubicomp area [88].

Intuition: The consistency pattern defined in [88] captures the requirement that some given event Q always causes some effect R (expressed as a query over the states before and after Q). There is optionally an additional query (a guard) that constrains when the system behaves consistently.

In the ubicomp context: the event (Q) is either an explicit or an implicit action by the user. It might also be a change in the environment, or an internal action of the system. Implicit actions and environment changes will be expressed in terms of the values read by the sensors in the system. The effect of the action (R) represents a change in the state of the system as a whole, which may be reflected in the environment itself as perceived by its users who must also be considered. Context plays an important role. If an action by some user is being analysed then the presence of other users might also influence the response. Hence, the gate in the library may not close when a user leaves its neighbourhood because another user might be close to it. All these dimensions provide a rich texture in which the pattern can be used beyond simply providing values for Q and R. The whole context of the environment being analysed is described by the tokens held by the *small colour sets* initialized by the APEXi tool.

Algorithm: Relevant states in terms of the SS tool are defined as those states in the reachability graph that satisfy the guard S (if defined, otherwise all states are considered). The algorithm considers all transitions in the reachability graph that both originate from relevant states and correspond to the action Q, and then asserts the query R over states that these transitions lead to. In this case the skeleton of the algorithm to be followed is presented in Figure

6.17. This skeleton is used to identify counter examples. The algorithm identifies firstly relevant nodes (corresponding to the effect R) for the property being verified in the reachability graph (*identifyRelevantNodes* function). Nodes correspond to states of the reachability graph. The function parameter (*obj*) should be provided in agreement with the scenario being used for the analysis (i.e. the variable *obj* must be of the type of the tokens present in the place (*PLACE)* analysed). From the identified relevant nodes (returned by the *identifyRelevantNodes* function) the algorithm identifies the nodes of the counter example (*counterExampleNodes* function) mapping the *counterExampleNodes* function to the set of relevant values of the selected scenario and holding the resulting list of nodes in a variable (*CONSISTENCY*). Concrete instantiations of this algorithm are presented in Chapter 7. The underlined pieces in Figure 6.17 are the parts that need to be instantiated. In summary, the application of this pattern after being instantiated implies firstly the evaluation of the *identifyRelevantNodes* function. Secondly the evaluation of the *counterExampleNodes* function that depends on the *identifyRelevantNodes* function and finally the evaluation of the predicate that provides the list of the counter example nodes.

6.3.2 The feedback pattern

Justification: Feedback is another common heuristic. A particular use of the consistency pattern is to verify that the system provides consistent feedback.

Intuition: The feedback pattern therefore is a specialisation of Consistency, where a user action Q always causes a perceivable effect R.

In the ubicomp context: the action (Q) must be either an explicit or implicit action by a user, or some internal change to the system state that the user must be made aware of. The effect of the action (R) represents a change in the environment as represented by observable features of the environment. It is important to recognise that the person causing the system's response might not necessarily be the same person as the person to whom the response is directed. Even if it is the same person, the fact that the response might be triggered by an implicit interaction or an environment change begs the question of whether the response will be salient enough. At this stage issues such as salience are not being considered, rather the concern is to guarantee that feedback is always provided. It is likely that evaluating the salience

of a particular feedback will require input from the simulation (an example of synergy between the formal and empirical analysis - but see [69] for a formal treatment of salience).

```
fun identifyRelevantNodes obj =
PredAllNodes (fn n => cf(obj,Mark.MODULE'PLACE 1 n) > 0)
--------------------------------------------------------------------
fun counterExampleNodes u =
 let
  val nodes = identifyRelevantNodes u
 in
  let
   val predecessorsNodes =
   SearchNodes (
   nodes,
   fn _ => true,
   NoLimit, fn n => InNodes n, [], op ^^)
  in
    let
     val nodesPredecessorsNodesWith2orMoreSucessors =
     SearchNodes (
     remdupl(predecessorsNodes),
     fn n => not(contains nodes  [n]) andalso
               length( OutNodes n) >= 2,
               NoLimit, fn n => n, [], op ::)
    in
     SearchNodes (
     remdupl(nodesPredecessorsNodesWith2orMoreSucessors),
     fn n => not( contains (map (fn x=> Reachable(n,x) andalso
              length(NodesInPath (n,x))>2) nodes) [true]),
     NoLimit, fn n => n, [], op ::)
    end
  end
end
--------------------------------------------------------------------
val CONSISTENCY =
         map(fn u => counterExampleNodes u)
                    (UpperMultiSet (Mark.MODULE'PLACE 1))
```

Figure 6.17: Consistency/feedback property algorithm skeleton

Algorithm: The algorithm considers all transitions in the reachability graph, from relevant states, that correspond to the action Q and then asserts the query R to the states that these transitions lead to. Relevant states in terms of the SS tool are defined as those states in the reachability graph that satisfy the condition S. The algorithm skeleton to be followed and instantiated is the same as defined for Consistency. The restriction to verify a feedback property with this algorithm is that the relevant nodes being analysed (returned by the *identifyRelevantNodes* function, see Figure 6.17) should cause a perceivable effect. Hence, *Feed-*

backAlgorithm(Q,R) = *ConsistencyAlgorithm*(Q,R) where Q always causes a perceivable effect R.

6.3.3 The reachability pattern

Justification: reachability is a basic property from which other properties are derived (e.g. precedence, completeness). It can be used to demonstrate that the system can reach a specific state or situation.

Intuition: The reachability pattern captures the requirement that the system can always evolve from one specific state S to another state Q.

In the ubicomp context: the environmental situation is represented as a state with particular distinguishing features, for example, the light associated with a book being switched on, or a user being at a given location. Hence, some features of the state are directly controlled by the system (the light in the book), while others are observed (the user's position). Nevertheless, the observed features might be indirectly influenced by the system (e.g., the gate, when it opens, enables the user to move inside the library). In a complex system, establishing how these influences work will not be easy, and performing verification will help in this respect.

Algorithm: The algorithm uses the reachability graph and identifies desired states with identifying attributes. For each identified state S the algorithm checks whether it is possible to reach a new state Q with the desired environment attributes. The associated algorithm skeleton is presented in Figure 6.18. It identifies all target nodes (state Q – *targetNodes* function). The token (*TOKEN*) and the place of the model where to search for (*MODULE'PLACE*), need to be in agreement. In a similar way original nodes (state S – *originalNodes* function) are identified. From each node of the original list the algorithm identifies the nodes from which it is impossible to reach a node of the target list (the counter examples - held in the *REACHABILITY* variable). If the returning list of nodes is empty this means that for each state S it is possible to reach a new state Q with the desired environment attributes. Concrete instantiations of this algorithm are presented in Chapter 7. The application of this pattern after being instantiated (underlined parts of the model) implies two evaluations. Firstly the evaluation of the *targetStates* and *originalStates* functions. Secondly the evaluation of the *SearchNodes* function (Figure 6.18), that depends on the values returned by the *targetStates* and *originalStates* functions (variables *TS* and *OS*).

```
fun targetNodes obj
= PredAllNodes (fn n => cf(obj,Mark.MODULE'PLACE 1 n) > 0)

val TS = targetNodes TOKEN
--------------------------------------------------------------------------------
fun originalNodes obj
= PredAllNodes (fn n => cf(obj,Mark.MODULE'PLACE 1 n) > 0)

val OS = originalNodes TOKEN
--------------------------------------------------------------------------------
val REACHABILITY =
  SearchNodes (OS,
  fn n =>  not ( contains (map (fn x=> Reachable(n,x)) TS)) [true] ),
  NoLimit, fn n => n, [], op ::)
```

Figure 6.18: Reachability property algorithm skeleton

6.3.4 The precedence pattern

Justification: The precedence pattern describes the relationship between a pair of events/states where the occurrence of the first is a necessary pre-condition for the occurrence of the second.

Intuition: This pattern captures the requirement that a state or event S precedes another state or event P. The occurrence of the second is enabled by the occurrence of the first.

In the ubicomp context: This property can be used to verify that some event or state does not occur without the satisfaction of a pre-condition. Consider for example the stated property concerned with the illumination of a book light. The first state (S), triggered by a user action (for example, the user approaching the book), is a pre-condition for the occurrence of the second state (P – book light turning on). In this example the property requires that the light will never turn on without a relevant user approaching the book. Note that this does not guarantee that the light will always turn on when a relevant user approaches the book. That is expressed by the consistency pattern.

Algorithm: The algorithm identifies the target states (P) of the reachability graph and then verifies the presence of the original state S in the list of predecessors states of P. The algorithm skeleton is presented in Figure 6.19. It identifies all target nodes (state P – *targetNodes* function) looking for tokens in a specific place (e.g. book with id 1 in the *lightedBooks* place). Analogously the same approach is taken to identify the original nodes (state S - *originalNodes* function). Finally for each node of the target nodes' list the algorithm identifies all

predecessor nodes and check whether they are not members of the original nodes list (counter examples - hold in the *PRECEDENCE* variable). If the returning list of nodes is empty this means that each state P is preceded by a state S.

The underlined pieces in the algorithm are the part that need to be instantiated. The token (*TOKEN*) to look for, and the place in the model that need to be searched (*MODULE'PLACE*), must be provided. The application of this pattern after being instantiated is accomplished by evaluating the *targetNodes* and *originalNodes* functions and subsequently the *searchNodes* function which depends on the returned values from the *targetNodes* and *originalNodes* functions (*TN* and *ON* variables).

```
fun targetNodes obj
= PredAllNodes (fn n => cf(obj,Mark.MODULE'PLACE 1 n) > 0)

val TN = targetNodes TOKEN
-------------------------------------------------------------------------
fun originalNodes obj
= PredAllNodes (fn n => cf(obj,Mark.MODULE'PLACE 1 n) > 0)

val ON = originalNodes TOKEN
-------------------------------------------------------------------------
val PRECEDENCE =
   SearchNodes (
   InNodes TN,
   fn n =>  not(contains ON n),
   NoLimit, fn n => n, [], op ::)
```

Figure 6.19: Precedence property algorithm skeleton

6.3.5 The completeness pattern

Justification: this enables the determination of all system states that can be reached from any state in the system in only one action. This property may be useful to check if features of the environment are always easily accessible to users. By easy we mean here that they are achieved in just one action.

The intuition: is to verify that a user can get from a state S to any other state in one step.

In the ubiquitous context: A new state is identified by a change of the environment attributes and the change is triggered through one explicit or implicit action. Who made the action or satisfied the condition is not relevant. The focus here is to determine that given any

situation in the environment it is possible to move to all other possible situations/states in one step (e.g. user action, change of context).

Algorithm: The state S is represented in the reachability graph. The algorithm skeleton present in Figure 6.20 is used to verify the property. It checks whether the successors of the state S contain all possible system states. The variable *ALLNODES* holds all nodes of the reachability graph. The *propertyHoldsNODES* variable holds all nodes that are reachable from S (*NODE_S* variable) in only one action. Finally the variable *COMPLETENESS* indicates if the property analysed is verified (number of nodes of the reachability graph should be equal to the number of nodes reached in only one step from S) for the state analysed. The algorithm skeleton should be instantiated (underlined parts) with the node representing the state S.

Thimbleby and Oladimeji [90] argue that completeness is a way of measuring usability of interactive systems. However being able to accomplish everything in just one action can of course lead to low levels of usability. Nevertheless this pattern provides the possibility to check the completeness of the system.

```
val ALLNODES =
  SearchNodes (EntireGraph,fn n => true, NoLimit, fn n => n, [], op ::)

val propertyHoldsNODES =
  SearchNodes(EntireGraph,
              fn n => Reachable(NODE_S,n) andalso length(NodesInPath (NODE_S,n)) <= 2),
              NoLimit,
              fn n => n,
              [],
              op ::)

val COMPLETENESS =
   (length ALLNODES) = (length propertyHoldsNODES)
```

Figure 6.20: Completeness algorithm skeleton

6.3.6 The reversibility pattern

Justification: undo is a very important feature of most systems to provide the means of undoing mistakes (wrong actions). The property is important to prevent user frustration.

Intuition: This property requires that the effect of action Q on attributes in S can be undone eventually. The intention is to check whether the effect of an action can eventually be undone.

In the ubicomp context: some actions are implicit and consequently it can be difficult to undo them. One feature associated with this property is that the action performed (Q) should provide feedback in the environment. If this does not occur then it is difficult for a user to decide whether the effect is undone. Additionally it is difficult for users to undo the effect of an action if they do not know what was the cause of the effect. Indeed because many users can interact in the environment an effect can be caused by another user. This property pattern can be verified by an algorithm but this does not mean that users know how to undo the effect. Characteristics such as these are the focus of the user experience analysis provided by the 3D simulation of APEX.

Algorithm: If we were dealing with an undirected graph the verification of this property would be immediate. However the reachability graph is a directed graph and consequently to verify the property we check that from a desired state an action leads to a new state and the previous state can be reached. This can be done by finding all the states reached by executing Q and applying the reachability pattern from that state to the state before Q. Figure 6.21 shows the algorithm skeleton to identify all states reached by executing Q (*OUTNODES* variable). To verify a reversibility property the nodes before Q (*AQN*) are provided as the target nodes of the reachability pattern and the nodes after Q (*OUTNODES*) are provided as the original nodes of the reachability pattern. Analogous to previous patterns this algorithm needs to be instantiated (underlined parts).

```
fun actionQNodes obj
= PredAllNodes (fn n => cf(obj,Mark.MODULE'PLACE 1 n) > 0)

val AQN = actionQNodes TOKEN

val OUTNODES =
  SearchNodes (outNodes AQN, fn n => true, NoLimit, fn n => n, [], op ::)
```

Figure 6.21: Algorithm skeleton for the identification of the nodes before and after Q

6.3.7 The possibility pattern

Justification: The intent of this property is to express that some event or state is always possible throughout the execution of the system. Note that it does not require that the state or event actually happens in a specific execution of the model, only that it is possible that it will. This property can be used to identify features of the system that cannot be accessed.

Intuition: this property pattern means that an event or state P is always possible to be accessed.

In the ubicomp context: consider an ubicomp system that sends alerts, this is crucial to know independently of the system state if it is always possible to reach the alert state P.

Algorithm: the reachability graph provides all possible states that can be reached by the execution of the system. The algorithm checks if for every state we can reach the state P. This can be done by applying the reachability pattern from all states (original nodes in the reachability algorithm) to P (target node in the reachability algorithm). The function of Figure 6.22 provides every node of a reachability graph (*ALLNODES* variable).

```
val ALLNODES =
  SearchNodes (EntireGraph,fn n => true, NoLimit, fn n => n, [], op ::)
```

Figure 6.22: Function to provide all nodes of the reachability graph

6.3.8 The universality pattern

Justification: this property pattern aims to demonstrate that a property is always true throughout the system execution. Consider for example that we want to verify that the number of objects in the environment remains constant throughout the system execution.

Intuition: the intent presented by Campos et al. [87] is that some event or state occurs in every state of the execution of the system.

In the ubicomp context: In the example given above this means that the number of objects (attributes) in any state of the system possesses always the same value. This relates to the interpretation of patterns presented in Section 6.1.2 in particular to what is being analysed. In this example the property is more focussed to validate the model itself instead of addressing the system's design.

Algorithm: each state of the reachability graph is analysed, checking whether their attributes satisfy the property. The Figure 6.23 shows the algorithm skeleton to be used. The variable *ALLNODES* holds all nodes of the reachability graph. The *propertyHoldsNODES* variable holds all nodes where the property is true. Finally the variable *UNIVERSALITY* indicates if the property analysed is always true for every state (number of nodes of the reachability graph should be equal to the number of nodes where the property is true). The underlined part of the figure should be instantiated to identify the nodes that hold the property being verified.

```
val ALLNODES =
  SearchNodes (EntireGraph,fn n => true, NoLimit, fn n => n, [], op ::)

val propertyHoldsNODES =
  SearchNodes (EntireGraph,fn n => PROPERTY(n), NoLimit, fn n => n, [], op ::)

val UNIVERSALITY =
  (length ALLNODES) = (length propertyHoldsNODES)
```

Figure 6.23: Universality property algorithm skeleton

6.3.9 The eventually pattern

Justification: the need to demonstrate that a state or event can become true.

Intuition: this property pattern means that a state or event P eventually becomes true as stated by Campos et al. [87].

In the ubicomp context: this property pattern is used to identify whether an undesired behaviour can occur. For example in the case of the book light example, whether the light can be on and off at the same time is a relevant property to be verified.

Algorithm: this property is verified by analysing each state of the reachability graph and identifying whether at least one state possesses the attributes that satisfy the property. Figure 6.24 presents the algorithm skeleton to be used. The *propertyHoldsNODES* variable holds all nodes where the property is true. The variable *EVENTUALLY* indicates if the property analysed is true. The number of nodes where the property is true is greater than zero. The underlined part of the figure should be instantiated to identify the nodes that hold the property being verified.

```
val propertyHoldsNODES =
  SearchNodes (EntireGraph,fn n => PROPERTY(n), NoLimit, fn n => n, [], op ::)

val EVENTUALLY =
  (length propertyHoldsNODES) > 0
```

Figure 6.24: Eventually property algorithm skeleton

In Chapter 7 some of these property algorithm skeletons are made concrete through examples by instantiating them. Results demonstrate that this approach is adequate for analysis while providing a uniform process for the identification and verification of properties.

6.4 Alternative Analysis

Besides state space analysis different techniques are also possible using the models with tailored queries. For example, CPN simulation together with state consultation can be used. The advantage of this simulation is that the model can be selectively executed step by step using an approach similar to program execution. This provides a way to see, by state consultation, how the system reacts in particular situations. This alternative form of analysis also serves as a complement to the exhaustive analysis. For example, when counter examples for some properties are identified, CPN simulation is used to recreate the identified situation and identify/understand the potential reasons for the failure. CPN simulations are provided with the CPN Tools and are automatically applicable to the developed models.

It is likely that evaluating a ubicomp environment by analysing its behaviour exhaustively does not guarantee that the proposed design solution provides an adequate experience. As seen with the feedback property pattern, a system satisfying this property could mean that at some level the system provides feedback but nevertheless the crucial elements of the environment that are actually required for feedback are missing from the analysis. It is impossible to know from this analysis if the feedback provided is salient, if it can be seen by the user or what it will look like physically. These are issues raised by analysis performed only at the modelling layer. However, as the APEX framework provides a multi-layered prototyping approach each layer supports a specific type of evaluation. Besides analysis of the model in the modelling layer, observation of virtual objects' behaviour, and user reaction, within a virtual environment are possible in the simulation layer and, observation of real objects con-

nected to the virtual environment, and users reaction to them in the physical layer. The analysis provided by these two additional layers are used as solutions to complement the analysis accomplished at the modelling layer.

6.5 Programmed Avatars

Besides formal analysis, the modelling layer supports the reasoning about the ubicomp environment being prototyped and provides features to make the simulation with several avatars simpler. APEX allows the creation of scenarios with programmed avatars. This is a different topic from providing values to the presence sensors in APEXi. Here the concern is to experience in the 3D simulation situations with several avatars and few or only one real user. When many users are involved simulations can be costly. The advantage of using programmed avatars is clearly identified when considering the human resources needed to experience large ubicomp environments where many users are interacting (e.g. an airport). With programmed avatars these costs are reduced because just one real user is strictly needed to evaluate the experience. Several programmed avatars were modelled that involved different navigations through the environment enabling one real user to experience the situations where many users are present. For example, the modelling of several programmed avatars arriving at a gate of the library at the same time makes it possible for the user to observe the behaviour of the gate when many users are within its proximity.

Avatars are programmed by defining their movement (other forms of interaction are currently not supported). This movement can be defined manually or use previously recorded information of real users exploring the simulation. Figure 4.5 in Section 4.3 shows the movement module of a programmed avatar.

Using programmed avatars the numbers of real users needed to analyse a scenario in the simulation is reduced as we can have a mixture of real users and programmed avatars. Alternating between non-programmed (driven by real users) and programmed avatars is easily achieved in APEX.

6.6 Conclusions

This chapter described the analysis of ubicomp environments made with APEX. The process used to accomplish the analysis was presented. It involves the use of the *analysis base model* and the closing of additional modules to make it tractable.

Because some system aspects can have an effectively infinite set of values (e.g. time) and some scenarios are not useful for some properties, adequate value selection criteria is critical to effective analysis. When infinite values are needed the exhaustive analysis of all possible executions is infeasible, being consequently necessary to select adequate scenarios for analysis. The APEXi tool was developed to increase the automation of the analysis process and to make the scenario selection process easier. Scenario creation is simplified by providing desired values to the APEXi tool, which is responsible for populating the *small colour sets*.

The SS tool is applied to the resulting model, creating the relevant reachability graph. Queries are written over this graph in order to verify desired properties. Property patterns were specified to help developers identify relevant properties and write queries over the reachability graph. The patterns make it easier to verify properties using APEX.

This approach is illustrated in the next chapter using a smart library and an aware home as contexts. The examples illustrate the choice of property patterns and how their associated algorithm skeletons are instantiated in each example and then checked within the APEX framework.

The proposed analysis approach considers the development of queries in the ML language using property patterns that originally used properties formally described in CTL. An alternative approach for analysis could be the writing of CTL queries in the CPN tool using the ASK_CTL library [91]. However ASK_CTL has some limitations. Standard fairness properties cannot be expressed in ASK_CTL. Our approach enables the verification of a wider range of properties directly in ML. Because ASK_CTL queries are written in the ML language it will be possible to use ASK_CTL with the proposed approach if it is so wished.

The advantages of using programmed avatars for analysis with APEX were described. Additionally, alternative analysis approaches through CPN simulation and physical device usage where identified. The analysis based on the modelling layer is adequate for exhaustive and formal analysis providing support to verify useful properties. These different analyses

complement each other, providing more complete analysis support for ubicomp environments.

This chapter aimed to introduce the reader to the concepts involved with the analysis of ubiquitous environments using APEX. Useful analysis results are presented using larger and more useful examples in the next chapter.

Chapter 7

Examples

Examples of ubiquitous systems are considered in this chapter to highlight different aspects of the APEX framework (e.g. experimentation and analysis). These examples show the validity of the resulting prototypes and the expressiveness and application domain of the tool. Parts of these examples have already been used to introduce concepts.

7.1 Smart Library

The first example is a *smart library* where books are identified by means of RFID (Radio Frequency Identification) tags, and stored on bookshelves. Screens are used to provide information to library users. A registered library user is allowed entry or exit via gates. When registered users arrive at the entry gate, a screen displays which books have been requested by them earlier (possibly using a web interface at their desktop) and opens the entry gate.

The system recognises the user's position in real-time through presence sensors. It guides the user to the required books by means of screens or the user's PDA. As the user approaches a requested book's location a light with a specified colour is turned on. Several users who are looking for books in nearby locations can thereby distinguish their own request. When the book is removed or a certain time elapsed, the light on the book is turned off. As the user returns to the exit gate a personalised list of requested and returned books is displayed on a screen by the gate. The gate is then opened so that the user can leave.

This example is not highly problematic and can be considered somewhat anachronistic assuming a future of paper books on shelves. However while the example is not based on any specific existing system, systems similar to this can be envisaged to support dispatch in rela-

tion to e-shopping or for guiding people inside buildings (e.g. hospital or airport). Indeed, a method and system for localizing objects among a set of stacked objects equipped with improved RFID tags has been patented [92] suggesting the feasibility of the physical implementation of the system.

A system like the proposed could be used as part of a “chaotic storage system”. A chaotic storage system stores different types of items without allocating fixed positions to specific item types in order to maximize the usage of storage capacity. The *Bosch* factory in Braga[43] uses this system to store components between different phases of the production process. The existing working process is divided into two working spaces where the second space needs the material produced by the first space. Because of the vast diversity of products that are produced, a chaotic storage space is used between the two phases of the production process. This means that when workers need to store or collect specific items they need to determine where they should be/are stored and locate that position. A similar solution to the one used in the library could be used to locate and retrieve materials.

In the case of the library example a number of interesting issues can be explored. For example if 10 or 20 people look for books located on the same bookshelf at the same time it may be difficult for each to distinguish their own colour. Additionally the flashing of colours may disturb people as they are reading. Technological issues may also be explored in the prototype. For example, exploring different configurations of a virtual sensor in the virtual environment could help in the selection of a physical sensor to be used in the physical ubicomp environment. The proposed solution for locating books in a library is not claimed here as a perfect solution. The focus of concern is to explore the potential of APEX to enable the exploration of alternative design solutions and to identify issues that could be difficult or impossible to detect at a modelling layer. Additionally, the verification of properties of the behavioural solution is also considered.

7.1.1 The model

Creating the prototype involves, in addition to creating the virtual environment, extending the *CPN base model* provided by APEX (Appendix A) with the specific behaviour associated

[43] Bosch - Braga: http://www.bosch.pt (last accessed: 3 January 2012)

with the particular ubiquitous environment. The resulting model is then used to drive the virtual environment so that it can be used for evaluation as users explore the space. It is also the basis for verification. This section presents two modules that must be added for this example (the gate modules and the PDA module that directs the user).

In the smart library a number of modules are designed to simulate the behaviour of gates, books, PDAs and displays. The *Gate* module is described using CPN in Figure 4.2 (Section 4.2.1, page 61) and holds information about the users, the devices and relevant presence sensors in the environment. As stated in Section 4.2, the purpose of the gates module is to open a gate when a user, who has appropriate "entering" permission, is in the proximity of a presence sensor associated with the gate. The *Gate* module consists of a transition to open a gate and another one to close it. Whether the module will open or close the gate is based on information held in a number of places: *Dynamic Objects* (i.e. gates), *Users* and *P_sensors* (presence sensors). When executed the transitions carry out their associated action: *sendOpenGate* function in the case of the *open gate* transition, and *sendCloseGate* function in the case of the *close gate* transition. These functions are responsible for sending the identity of the gate that is to be opened or closed to the simulation layer.

The opened or closed state of the gates is recognized in two places: the *Dynamic Objects* place (holds tokens for closed gates) and the *gates opened* place (holds tokens for opened gates). The *is* function is used to identify, in the *Dynamic Objects* place, what type of objects we are dealing with, particularly here to identify the gates. This is necessary because the *Dynamic Object* place holds dynamic objects other than gates (e.g. screens, books). The distinction is based on the token's information type that the tokens hold (e.g. gate, screen). The function is designed to receive a *Dynamic Object* and a string as arguments, and to compare the type of the object against the string to check whether there is a correspondence. After the token is identified as being a gate, it is necessary to get further information to decide whether to open or close the gate. Relevant information is required to ascertain if:

1. a user is near a presence sensor;
2. the presence sensor affects the gate;
3. nobody is near the presence sensor.

Three functions were developed to capture these conditions, respectively:

1. *userNearPresenceSensor;*
2. *objAfectedByPresenceSensor;*

3. *nobodyNearPresenceSensor*.

Using them the behaviours of the gates are modelled. A module responsible for providing directions to users to reach the books they have requested was also added. This is presented in Figure 7.1. Using both the user's position and the position of the desired books, the *getDirection* function (associated with the *show direction* transition) returns the direction to be followed (e.g. North, North East, South, East, etc.). This information is then forwarded to each avatar's PDA connected to the simulation (*sendUserInfo* function of the *show direction* transition). This information is interpreted by the APEX *communication/execution component* and provided to avatars in possibly different ways (e.g. text, arrows, voice). When the direction is sent to a physical PDA and arrows are used, the application running on it must orientate the direction correctly, for example, a *southwest* direction obtained from the model (originated from the *show direction* transition) must be converted to an arrow pointing in the appropriate direction. Other important modules developed include one responsible for picking up books and notifying relevant users.

Figure 7.1: User's PDA book direction module

Once all specific modules are connected to the *base model* they animate the library environment that is to be explored by users as described in Section 4.2.3.

For the purposes of exhaustive analysis the modules must be added to the *analysis base model* (Appendix B), and the analysis process carried out as explained in Section 6.2.

7.1.2 Instantiating property templates

Having obtained an appropriate model, it is now possible to proceed with the analysis. This complements the results obtained by evaluating the users' experience of the prototype. Analysts may know which property they want to prove (e.g., by observing real users as they interact with the simulation), but they can also have difficulties in identifying them appropriately. Property templates help them in this task. By capturing (and thus guiding the analysis towards) potentially relevant features of a design, they help the analyst discover appropriate properties. Additionally, using property algorithm skeletons makes it easier to verify properties because queries used in instantiating each of the property patterns can be reused. Three property patterns are considered in relation to this example: feedback, reachability and precedence (their description can be found in Section 6.3). This section shows how the templates help the identification of properties (associated with patterns). It starts by presenting what the pattern is about and provides the means to reach a specific property for analysis based on a situation relevant to the example. The next section focuses on showing how algorithm skeletons help in the verification of identified properties and presents the process that is involved.

Feedback

This pattern is used for the identification and verification of relevant feedback properties. It is used to capture the requirement that some given event causes an effect. In this concrete example feedback is a relevant property. For example, one specific situation where feedback is relevant is when users approach their wanted books and the light associated with the book should be turned on. The feedback template describes this situation formally as a generic property. The pattern facilitates verification through the associated algorithm skeleton (as will be described in the next section). Parameters for property templates are instantiated with user interactions, environment changes and features or states of the environment. In this case the feedback variables were instantiated with the following values:

- action (Q): defined as the implicit action that occurs when the user approaches the book. The proximity of the user to the book is detected by presence sensors in the environment;
- effect (R): defined as changing the environment so that the relevant light is switched on;

- guard (S): defined as stating that the light must initially be off for this property to hold.

These particular variable assignments identify and formally describe the stated situation as being a feedback property. The following property is identified: "*when a user approaches the appropriate bookshelf the book lights up (unless it is already on)*". However there are probably other variable assignments that do not produce interesting or meaningful properties. This template requires variables to be assigned to help identify the particular feedback property but the interest or usefulness of the resulting property is not guaranteed and depends on the selected values.

Reachability

This pattern is used for the identification and verification of reachability properties. It is used to capture the requirement that the system can always evolve from one specific state to another specific state. In this concrete example reachability might be relevant for example when a user should be notified that the book being looked for has been picked up by another user. The reachability template provides two variables (initial state Q and target state S as explained in Section 6.3) to be assigned. The following values can be used to describe formally the situation as being a reachability property:

- state Q: a book that a user is looking for is picked up by another person (stops being available);
- state S: the user is notified.

These variable assignments identify the situation as the following reachability property: "*If a book that a user is looking for is picked up by another person (stops being available), it is possible to notify the user*".

Precedence

This pattern is used for the identification and verification of relevant precedence properties. It is used to capture the requirement that the occurrence of a state or event is enabled by the occurrence of a first state or event. In the example this type of property might be relevant. For example, one specific situation where precedence is relevant is that a book light should not

turn on while the user desiring it is not close to it. The precedence template variables (state S and then state P) are assigned with values relevant to the precedence property:

- state S: relevant user is near the bookshelf;
- state P: the light turn on.

These variable assignments identify the following property "*the light does not turn on while the relevant user is not near the bookshelf*".

7.1.3 Checking the model using the SS tool

As stated in Chapter 6 APEX provides support for analysis. The SS tool plays a crucial role in the checking of properties. The use of the property patterns related to the properties in the previous section is now considered. This section illustrates the verification of useful properties of the smart library example through the instantiation of property skeletons.

Feedback

The developer should select a scenario with adequate values for the desired analysis as illustrated in Section 6.2. The scenario that is used as a basis for analysis using the feedback property "*when a user approaches the appropriate bookshelf the book lights up (unless it is already on)*" uses the following selected values provided to APEXi (see Figure 7.2):

- one user (*Test User*) who desires the book with identifier number 1 (*values* field);
- two presence sensors, one at the entrance and the other near the bookshelf. The sensors affect the gate and the book's light (*Sfeatures* field) respectively;
- two books with identifiers 1 and 2 (*OBJfeatures* fields);
- one gate with identifier 3 (*OBJfeatures* fields).

Using the feedback pattern, the information that is required includes the identification of all possible paths that the models might take when reacting to the user approaching the bookshelf. These paths would be explored to see if the desired behaviour holds. At any moment many different events might be triggered in the specified environment. When the user approaches the bookshelf, depending on how the model is developed, it might react immediately, or might delay the reaction until it has finished reacting to other events. From a sys-

tem's design perspective, what needs to be guaranteed is that after all relevant events have been processed the book light will be turned on.

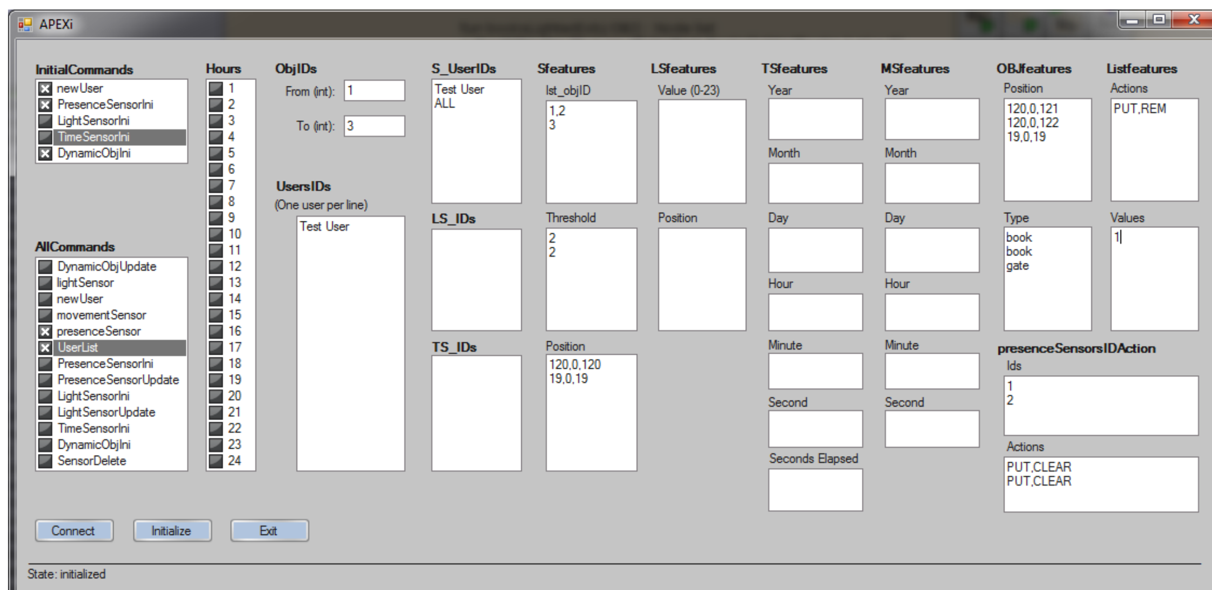

Figure 7.2: APEXi - selected values

For this purpose the algorithm skeleton of this pattern (see Figure 6.17) was instantiated (see Figure 7.3), turning the algorithm into concrete functions and trying to identify a counter example. The *identifyRelevantNodes* function of the algorithm skeleton was instantiated with the place where to search (i.e. *Books'LightedBooks*) to identify the relevant nodes (see Figure 7.3). This function identifies the desired nodes of the reachability graph i.e., the ones that provide the desired feedback. In this case the nodes are the ones that have books with lights switched on. The other generic part of the algorithm skeleton was also instantiated (underlined part of Figure 6.17), the place *AnimationSetup'Dynamic_Objects* used to identify the nodes to be used in the analysis, arguments of the *counterExampleNodes* function (see Figure 7.3).

After being instantiated, this concrete algorithm identifies, in this case, those nodes where the user is near the desired book and the book's light has not turned on (*FEEDBACK* variable in Figure 7.3). The algorithm identifies firstly the nodes in the reachability graph where the user is already detected near the bookshelf, but the system is still to react. From these nodes the system can either turn the book's light on, or alternatively choose to process some other relevant event. Selecting the second alternative (doing something else), creates

the executions from which a node with the book's light on is not reached (*counterExampleNodes* function). The resulting list of nodes is empty. This means that for the analysed scenario (considering the values provided) there is no system execution containing a node where the light should be turned on but was not.

Summing up, the feedback property algorithm skeleton was used to verify the property template. As stated the instantiation is simply accomplished, in this case, by indicating the places where to search for the relevant nodes used by the algorithm.

```
fun identifyRelevantNodes obj =
PredAllNodes (fn n => cf(obj,Mark.Books'LightedBooks 1 n) > 0)
--------------------------------------------------------------------------------
fun counterExampleNodes u =
 let
  val nodes = identifyRelevantNodes u
 in
  let
   val predecessorsNodes =
   SearchNodes (
   nodes,
   fn _ => true,
   NoLimit, fn n => InNodes n, [], op ^^)
  in
   let
     val nodesPredecessorsNodesWith2orMoreSucessors =
     SearchNodes (
     remdupl(predecessorsNodes),
     fn n => not(contains nodes  [n]) andalso
              length( OutNodes n) >= 2,
              NoLimit, fn n => n, [], op ::)
   in
     SearchNodes (
     remdupl(nodesPredecessorsNodesWith2orMoreSucessors),
     fn n => not( contains (map (fn x=> Reachable(n,x) andalso
             length(NodesInPath (n,x))>2) nodes) [true]),
     NoLimit, fn n => n, [], op ::)
   end
  end
end
--------------------------------------------------------------------------------
val FEEDBACK =
          map(fn u => counterExampleNodes u)
                   (UpperMultiSet (Mark.AnimationSetup'Dynamic_Objects 1))
```

Figure 7.3: Book's light behaviour property (concrete feedback algorithm instantiation)

The property "*whenever the designated user moves away the light turns off*" is another feedback property relevant to this example and its demonstration follows the same procedure but using a different instantiation of the algorithm skeleton. The instantiation is accomplished indicating the places where relevant nodes can be identified (in this example these are when

the light is off). This property demonstrates a different feedback property that concerns how the light turns off.

Reachability

The property, "*If a book that a user is looking for is picked up by another person (stops being available), it is possible to notify the user*" is now addressed. This is a reachability property, and its verification thus follows the algorithm skeleton associated with the reachability pattern (see Figure 6.18). In this case this property verifies if, for every user looking for the same book and every time the book is picked up a user notification state is reachable.

As in the previous example, this property is verified using a specific scenario. The scenario is composed of books, presence sensors and three users (*Silva, Rodrigues* and *Cardoso*) looking for the same book (with identifier equal to 1). The property algorithm skeleton should be instantiated, considering the selected values for the *small colour sets* specified in APEXi. For example, in this instantiation (Figure 7.4), the user *Silva* and the book with *id* equal to 1 are values of the parameter (*userIDxOBJ*) used because these are elements which compose the scenario selected for analysis (APEXi selected values).

The idea behind the demonstration of this property is to identify states from which a user picks up a book and the system is not able to reach a notification state for users looking for this book. In other words the aim is to find counter examples where the system does not have the required properties. Figure 7.4 shows the instantiation of the reachability pattern algorithm skeleton (see Figure 6.18). This is achieved by instantiating the *targetNodes* and *originalNodes* functions to identify the relevant nodes (see underlined pieces in Figure 7.4). The places (i.e. *BookPickUp'User_Notified* and *BookPickUp'OBJ_deleted*) used to identify the nodes to be used in the analysis and concrete tokens (i.e. *userIDxOBJ* and *book*) to be identified in these places are provided. By this means the desired property can be verified.

The execution of this concrete algorithm identifies firstly all notification nodes (returned by the *targetNodes* function). When these have been identified all nodes in which the book is picked up are identified (returned by the *originalNodes* function). The final stage is to identify any node in which the book is picked up and from which no notification can be made, i.e. no notification node is reachable (hold in the *REACHABILITY* variable). Checking this property using the skeleton (with each of the three users as parameter) returns no nodes (*REACHABILITY* variable value) which means that for the selected scenario (three users looking for

the same book) whenever a user picks up a book it is possible to notify all users looking for the book. For the selected scenario this property is verified.

```
val userIDxOBJ =
("Silva",{id="1",objType="book",position={x=120,y=0,z=121}}) : USERIDxOBJ

fun targetNodes obj
= PredAllNodes (fn n => cf(obj,Mark.BookPickUp'Users_Notified 1 n) > 0)

val TS = targetNodes userIDxOBJ
-------------------------------------------------------------------------------
val book =
{id="1",objType="book",position={x=120,y=0,z=121}} : OBJ

fun originalNodes obj
= PredAllNodes (fn n => cf(obj,Mark.BookPickUp'OBJ_deleted 1 n) > 0)

val OS = originalNodes book
-------------------------------------------------------------------------------

val REACHABILITY =
  SearchNodes (OS,
  fn n =>  not ( contains (map (fn x=> Reachable(n,x)) TS)) [true] ),
  NoLimit, fn n => n, [], op ::)
```

Figure 7.4: Notification property (concrete reachability algorithm instantiation)

Precedence

The third property "*the light does not turn on while the relevant user is not near the book*" follows the precedence property pattern. To reach a state where the light is on, a relevant user must be near the book. The instantiated precedence algorithm skeleton consists in firstly identifying the nodes where the light is on (*targetNodes* function instantiation, Figure 6.19) and secondly analysing their predecessors to check the presence of a user close to a book (*originalNodes* function instantiation, Figure 6.19). In the same way as the reachability pattern, the instantiation of the algorithm involves providing values to the functions to identify the relevant nodes (i.e. *MODULE'PLACE* and *TOKEN*, Figure 6.19) in agreement with the scenario used. The return of zero nodes means that for the selected scenario the property is always true.

APEX through CPN provides a way to analyse formally every portion of the system behaviour. Depending which property is to be proved different patterns were used and associated algorithm skeletons instantiated and executed. Patterns here help developers verify identified properties and to set up the queries used to verify them.

7.1.4 The prototype

Figure 7.5 shows the *smart library* prototype that was created. Interacting with it, users can experience proposed design solutions. For instance Figure 7.5 shows the book lighting system when users are near their desired books. Some issues are identified and solutions provided. For example, when evaluating the prototype, some users might feel that the book's light turned on too late i.e., when the user was too close to the book. By experiencing proposed solutions users are able to decide more accurately what the best distance between user and book is from which the book light should turn on. Without this experience it is more difficult to know which is the best value. Another issue resulting from user experience is presented in Figure 7.6. This figure shows the first view of a person reading a book at a table. It was recognised that the lights illuminated by the book localization system disturbs other readers in the same vicinity. This indicates that different solutions should be explored. Improving the precision of the guiding system could provide an effective alternative design.

Figure 7.5: Book's lights system

The exploration of a prototype using a 3D simulation provides clear advantages when prototyping ubicomp environments by allowing users to experience the system. This level of analysis complements the benefits provided by the modelling layer.

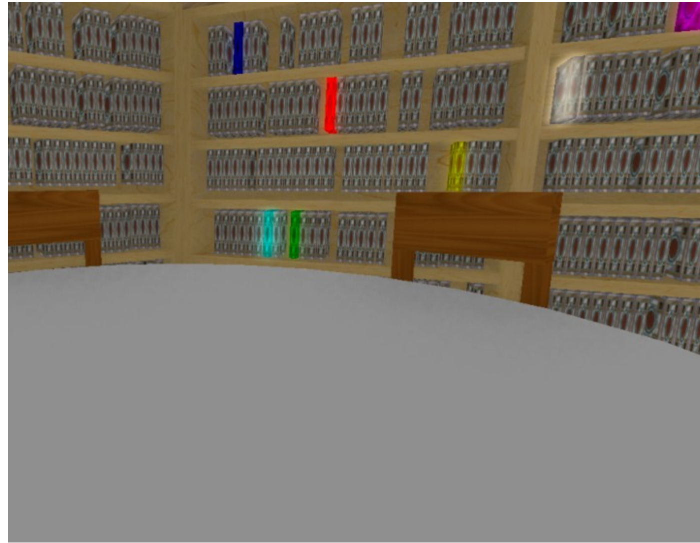

Figure 7.6: People's reading view

7.2 Aware Home

The second example aims to enable the exploration of new characteristics of APEX. In particular the example demonstrates the requirement for inter-user interaction and for the use of different types of sensors. The aim of the proposed system, an aware home, is to improve the quality of life and health education of child asthma sufferers. Asthma is a problem worldwide, with around 300 million affected individuals [93]. The U.S. Heart Lung and Blood Institute[44] claims that teaching patients and their family management skills improves asthma management and reduces the use of emergency services. Cabana et al. [94] indicate that parents need support in taking appropriate action to help asthmatic children who have difficulty identifying what triggers the asthma. Children spend the majority of their time indoors (e.g. home). Government and non-government organizations (e.g. Mothers of Asthmatics[45]) have developed home environment checklists but these lists provides general information and do not reflect the individual's real home environment [95]. The proposed system notifies parents when their child "approaches" a trigger and offers suggestions about how to act. This information is sent to a parent's mobile device or to a relevant screen in a room where the parent

[44] National Heart Lung and Blood Institute Diseases and Conditions Index: http://www.nhlbi.nih.gov (last accessed: 4 January 2012)
[45] Allergy and Asthma Network Mothers of Asthmatics: http://www.aanma.org (last accessed: 4 January 2012)

is located. Triggers are specific to individuals. They occur when relevant conditions in the environment are met (e.g. occurrence of tobacco smoke, house dust mites, pets, mould, outdoor air pollution, cockroach allergens) [96].

The prototype was made concrete by modelling a particular physical environment, the Aware Home[46] at Georgia Institute of Technology (GaTech). This home contains two identical floors with nine rooms each designed to explore emerging technologies and services in the home. The virtual environment is designed to represent the space, the sensors and the people within the space. The asthma system was developed to contain 16 presence sensors to detect the location of people and 16 environment sensors to detect environment conditions (for example smoke or air quality). Presence and collocated environment sensors are distributed across rooms as shown by the numbers in Figure 7.7.

Figure 7.7: Aware Home floor plan (without furniture) with inserted sensors (one presence sensor and one environment sensor present in each number)

3D modellers at GaTech had already developed a Google Sketchup 3D model of the Aware Home and further objects, representing furniture, were selected from an on-line 3D warehouse[47]. The resulting virtual environment is presented in Figure 7.8.

7.2.1 The model

The aware home is designed to alert parents when their child is in danger when particular environmental conditions have been met. This system is modelled using two modules de-

[46] Aware Home: http://awarehome.imtc.gatech.edu (last accessed: 15 November 2011)
[47] 3D warehouse: http://sketchup.google.com/3dwarehouse/ (last accessed: 15 November 2011)

scribed in Figure 7.9 and Figure 7.10. They hold information about the users and the sensors present in the environment. The purpose of the first module is to alert parents when their child is in the proximity of an asthma trigger. The module contains one transition to alert parents and another to remove the alert. Whether the module alerts or removes alerts depends on information held by the places: *Users* and *P_sensors* (presence sensors). An example of how the alert works is illustrated by the condition of the *Alert Parent* transition (expression between square brackets). This describes how the system alerts the parent of a child as they approach a trigger related to a specific allergy. The presence sensor is modelled using the *userNearPresenceSensor* function. The transition associated with this function causes the update of parent information with an acknowledgment (meaning that parent is alerted), described by the *updateUserValues("PUT",u1,"ACK")* function. The *Alert Parent* transition execution moves one token into the *Parents Alerted* state. During this transition parents are alerted as modelled by the *sendUserInfo* function. When the child is no longer close to an asthma trigger, indicated by *not(userNearPresenceSensor(ps,u)),* the system removes the token from the *Child parents Alerted* state. The acknowledgment that had been sent is then removed from the relevant parents using *(updateUserValues("REM",u1,"ACK").*

Figure 7.8: Aware Home 3D environment

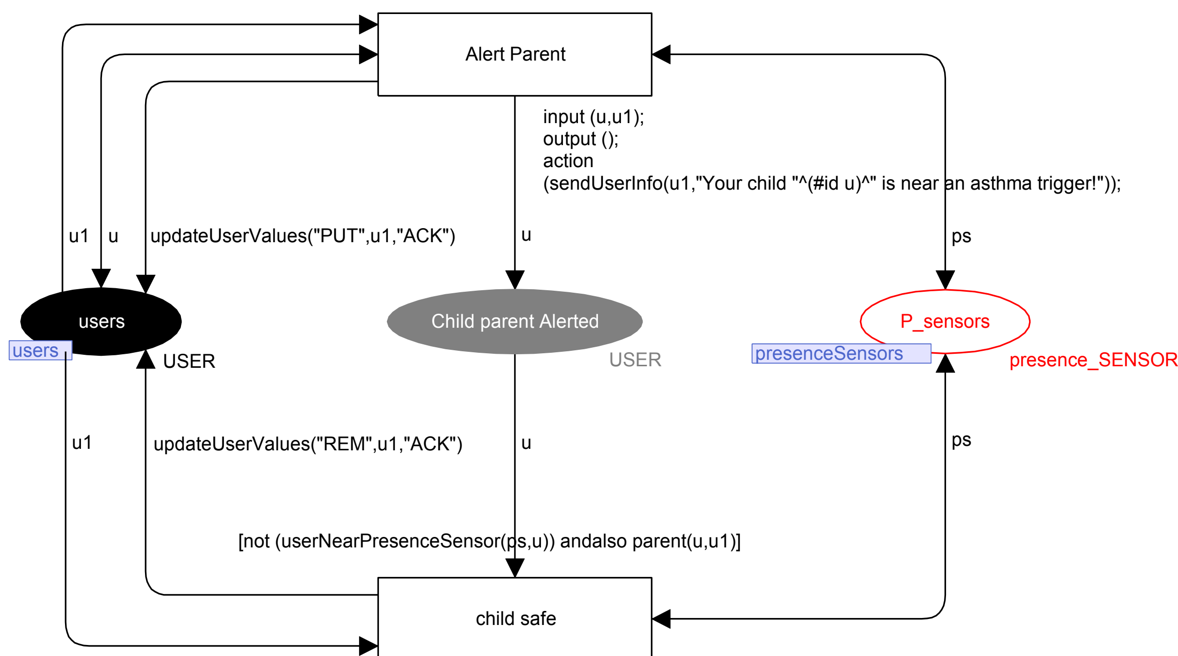

Figure 7.9: Parents' alert system behavioural model

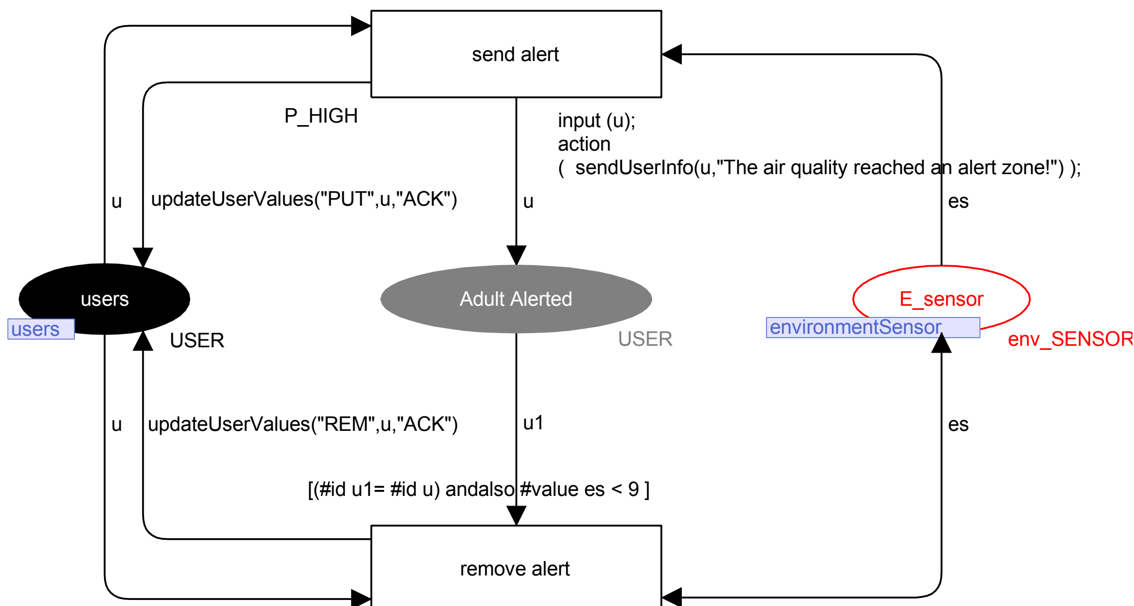

Figure 7.10: Air quality alert system

The other module that forms the model is responsible for sending an alert when the environment reaches an alert state (e.g. air polluted) as can be seen in Figure 7.10. The environ-

mental sensor (*E_sensor*) uses integers to indicate the environmental air quality. When the value is greater than or equal to 9 this means that an alert zone has been reached.

7.2.2 Instantiating property templates

As with the smart library example, having developed an appropriate model it is possible to proceed with the analysis. As before, this section shows how property templates help analysts in identifying and describing properties that will help in their verification. The next section shows how the properties identified in this section are verified by instantiating the algorithm skeletons associated with the property patterns. The same three property patterns used in the previous example are considered in relation to this example: feedback, reachability and precedence (see description in Section 6.3).

Feedback

This type of property might be relevant for this example. One specific situation were feedback is relevant is that when a child is in danger parents should be alerted. The variables of the feedback template were assigned with the following values:

- action (Q): defined as the implicit action that occurs when the child "approaches" an asthma trigger;
- effect (R): defined as changing the environment so that parents are alerted;
- guard (S): defined as stating that the parent must initially not be alerted for this property to hold.

These particular variable assignments identify and formally describe the stated situation as being a feedback property. The following property is identified: "*whenever a child is in danger parents are alerted*".

The following property "*whenever a child moves away from danger parents are alerted*" is another property formally described as being a feedback property by the variable assignment of this template with following values:

- action (Q): the child gets away from the asthma trigger;
- effect (R): parents are alerted;

- guard (S): the parent must initially be alerted for this property to hold.

Reachability

One specific situation where reachability is relevant is that parents can always receive information about their child wherever the child is. The variables of the reachability template assigned with the following values:

- state Q: wherever parents go;
- state S: parents can receive information about their child.

formally identify the following property: "*wherever parents go they can always receive information about their child*" as being a reachability property.

Precedence

An example of a precedence property is that a parent alert should always be preceded by a child being in danger. This will prevent false alerts, and promote parents' trust in the system. For this concrete situation the variables of the precedence template were assigned with the following values:

- state S: child in danger (close to asthma trigger);
- state P: parents are alerted.

This resulted in the identification of the following precedence property: "*the parents are not alerted if their child is not in danger*".

Instantiating the templates with actions, effects or conditions of the environment it was possible to identify new properties. In the next section these properties will be checked by instantiating the associated property algorithm skeletons and using the SS tool.

7.2.3 Checking the model using the SS tool

The identified properties are verified with the help of algorithm skeletons in the development of queries.

Feedback

The information needed to verify the property "*whenever a child is in danger parents are alerted*" is to identify all possible paths the models might take when reacting to the child approaching an asthma trigger and to see if the desired behaviour holds. At any moment many different events might be happening in the specified environment and when the child approaches the asthma trigger, the model might react to it immediately, or might delay the reaction until it has finished reacting to other events. What needs to be guaranteed is that after all relevant events have been processed the parents will be alerted.

Having identified the feedback property (see previous section) the associated property algorithm skeleton is instantiated and used to verify it. As before this instantiation is accomplished by providing to the *identifyRelevantNodes* function (see Figure 6.17) the place from which to search (*TriggersProximity'Child_parent_Alerted* - see Figure 7.11) to identify the target nodes to this situation. Additionally the instantiation of the parameter of the *counterExampleNodes* function, is provided by indicating the place (*AnimationSetup'users*) where the relevant tokens can be found.

After the algorithm has been instantiated it can be executed to verify the property. The *FEEDBACK* variable (Figure 7.11) holds those nodes where the child is near an asthma trigger and parents are not alerted. To identify these undesirable nodes which, if found, demonstrate that the property is not satisfied, the function identifies firstly the nodes in the reachability graph where the child is already detected near the asthma trigger, but the system is still to react. From these nodes the system can either alert parents, or alternatively choose to process some other relevant event. Selecting the second alternative (doing something else), the nodes from which a parent alerted node is not reached are identified (*counterExampleNodes* function of Figure 7.11). The resulting list of nodes is empty (*FEEDBACK* variable value). This means that for the analysed scenario (one parent, one child, one asthma trigger, one presence sensor and the simulation of movement of both parent and child in the home) there is no system execution containing a node where the parents should be alerted but were not.

Feedback (consistency) properties can be easily verified by instantiating the corresponding algorithm skeletons. A further example of a feedback property is "*whenever a child moves away from danger parents are alerted"*. This can be verified by instantiating the feedback algorithm skeleton. However, despite the verification of this property, parents may still want to check that their child is surely safe. In this context this property is not as useful as the

previous one, i.e. it is more important for the system to alert parents that their child is in danger, than to alert them that (s)he no longer is.

```
fun identifyRelevantNodes obj =
PredAllNodes (fn n => cf(obj,Mark.TriggersProximity'Child_parent_Alerted 1 n) > 0)
--------------------------------------------------------------------------------
fun counterExampleNodes u =
 let
  val nodes = identifyRelevantNodes u
 in
  let
   val predecessorsNodes =
   SearchNodes (
   nodes,
   fn _ => true,
   NoLimit, fn n => InNodes n, [], op ^^)
  in
   let
    val nodesPredecessorsNodesWith2orMoreSucessors =
    SearchNodes (
    remdupl(predecessorsNodes),
    fn n => not(contains nodes  [n]) andalso
             length( OutNodes n) >= 2,
             NoLimit, fn n => n, [], op ::)
   in
    SearchNodes (
    remdupl(nodesPredecessorsNodesWith2orMoreSucessors),
    fn n => not( contains (map (fn x=> Reachable(n,x) andalso
            length(NodesInPath (n,x))>2) nodes) [true]),
    NoLimit, fn n => n, [], op ::)
   end
  end
end
--------------------------------------------------------------------------------
val FEEDBACK =
        map(fn u => counterExampleNodes u)
                    (UpperMultiSet (Mark.AnimationSetup'users 1))
```

Figure 7.11: Parents alerted property (feedback)

Reachability

The property, "*wherever parents go they can always receive information about their child*" is verified by instantiating the reachability algorithm skeleton (Figure 6.18). The reachability property pattern checks whether it is possible, from one state, to reach another state (reachability between two nodes of the reachability graph). This is instantiated as "*for every parent position and every child position a parent alert state can be reached*". The algorithm skeleton associated with this pattern (see Figure 7.12) is instantiated in the same way as the reachability property of the smart library example. This is achieved by instantiating the *targetNodes* and *originalNodes* functions to identify the relevant nodes (see underlined pieces in Figure

7.12). The places (i.e. *Alert'parent_Alerted* and *Movement'ChildinDanger*) used to identify the nodes to be used in the analysis and concrete tokens (i.e. *parent* and *child*) to be identified in these places are provided. By this means the desired property can be verified.

The execution of the pattern identifies first all the parent's alerted nodes (returned by the *targetNodes* function). Then all nodes in which the child is in danger are identified (returned by the *originalNodes* function). Finally, the identification of any node from which an alert should be made, i.e. did not reach any parent's alerted node despite the child being in danger, are identified (hold in the *REACHABILITY* variable). The property did not return nodes which means that for the selected scenario wherever parents go they can receive alerts about their child. For the selected scenario (a child being in danger) this property is verified.

```
val parent =
{id="Silva", list=["adult","Rodrigo"]} : USER

fun targetNodes obj
= PredAllNodes (fn n => cf(obj,Mark.Alert'parent_Alerted 1 n) > 0)

val TS = targetNodes parent
--------------------------------------------------------------------------------
val child =
{id="Rodrigo", list=["child"]} : USER

fun originalNodes obj
= PredAllNodes (fn n => cf(obj,Mark.Movement'ChildInDanger 1 n) > 0)

val OS = originalNodes child
--------------------------------------------------------------------------------

val REACHABILITY =
  SearchNodes (OS,
  fn n =>  not ( contains (map (fn x=> Reachable(n,x)) TS)) [true] ),
  NoLimit, fn n => n, [], op ::)
```

Figure 7.12: Parents alerted property (reachability)

Precedence

The verification of the property "*the parents are not alerted if their child is not in danger*" follows the precedence property algorithm. The verification of this property is important in this example because it guarantees that parents are only alerted when their child is in danger. To reach a state where parents are alerted, the child must be near the asthma trigger. The precedence algorithm consists in firstly identifying the nodes of the reachability graph where parents are alerted (*targetNodes* function, see Figure 6.19) and secondly analysing their

predecessors (*originalNodes* function, see Figure 6.19) to check the presence of a child close to an asthma trigger (hold in the *PRECEDENCE* variable, see Figure 6.19). The return of zero nodes means that for the selected scenario the property is always true. Analogous to the previous examples the instantiation of the algorithm involves providing values to the functions to identify the relevant nodes (i.e. *MODULE'PLACE* and *TOKEN*, Figure 6.19) in agreement with the scenario used.

We have demonstrated that the suggested patterns can be instantiated to new examples. The property patterns (algorithm skeletons) identified describe a routine process on the reachability graph to identify states and demonstrate consequent properties. A summary of the routine processes of the property patterns verified in the examples are listed below:

- Check whether alternative concurrent transition executions hold the desired property (consistency property pattern);
- Check state reachability from a specific state (reachability property pattern);
- Check the presence of states holding desired values in specified state paths (precedence property pattern).

7.2.4 Checking non-functional properties

The physical features of the system are also highly relevant to users' experience of it. A different set of results can be obtained by applying queries over the reachability graph where physical features are key attributes. This is the focus of this subsection. Example queries would include demonstration that the positions of the sensors are adequate to drive users to books or that books are not present in an area that is not covered by a presence sensor. Notice that the verification of these properties is based on information not explicitly modelled at the behavioural level. The tokens of the model that represent the sensors are automatically updated with the position of the sensors in the virtual environment. An example demonstrating this type of property is presented in Figure 7.13.

The developed query demonstrates that for a provided scenario the distance between all sensors that compose the environment is less than two meters. The function *SensorHasOneSensorClose* achieves this distance analysis for the sensors present in the environment. The

function identifies the sensors that do not have another sensor close to them (less than or equal to two units in the example).

```
val sensors = UpperMultiSet (Mark.AnimationSetup'P_sensors 1)

fun distanceOk(ps:presence_SENSOR, ps1:presence_SENSOR, dist:real) =
 let val d =
   Math.sqrt(
         Math.pow(Real.fromInt(getPosX_PS(ps)-(getPosX_PS(ps))),2.0)+
         Math.pow(Real.fromInt(getPosY_PS(ps)-(getPosY_PS(ps))),2.0)+
         Math.pow(Real.fromInt(getPosZ_PS(ps)-(getPosZ_PS(ps))),2.0))
  in ( d <= dist)
end;

fun distanceAnalysis1(ps:presence_SENSOR, lst:presence_SENSOR list, l:INT, d:real) =
 if(l=0)
 then []
 else distanceOk(ps,List.nth(lst,l-1), d)::distanceAnalysis1(ps,lst,l-1, d)

fun distanceAnalysis(lst:presence_SENSOR list, dist:real, i:int) =
  if(i=length(lst))
  then []
  else distanceAnalysis1(List.nth(lst,i),lst, length(lst),dist)::distanceAnalysis(lst,dist,i+1)

fun EachHas2true(lst:bool list list) =
 if(length(lst)=0)
 then []
 else (contains_all (hd lst) [true,true])::EachHas2true(tl lst)

fun SensorHasOneSensorClose(sensors:presence_SENSOR list, dist:real) =
  let val a = distanceAnalysis(sensors,dist,0)
  in( EachHas2true(a))
end;

SensorHasOneSensorClose(sensors, 2.0)
```

Figure 7.13: Physical property (presence sensor distance)

This alternative analysis is also useful in the prototyping of ubicomp environments. Indeed the verification of non-functional features of the environment can be more important than functional ones. Imagine that analysts have verified, for the aware home example, that whenever a child is in danger parents are alerted. This verification can be useless if the presence sensors are not located close to the asthma triggers. These kinds of verification are the focus of this alternative analysis.

7.2.5 The prototype

The experience gained by using the APEX prototype resulted in the identification of some issues. For instance the way that alerts are communicated to parents can be improved. Some users prefer being notified by using fixed displays others by their PDA. However users do not always have their PDAs with them and consequently the model should consider this information to select how the alert is transmitted. Figure 7.14 shows a user experiencing a proposed alert system. The small window (bottom right corner) represents the user's PDA where in this case alerts are sent (a virtual PDA). APEX supports a multi layer approach and consequently if some resources are not available (e.g. a physical PDA) they can be executed in the simulation layer. Alternatively, as described in Section 3.1.1, physical devices can be connected through Bluetooth when available. Figure 7.15 illustrates the reception of an asthma trigger parent alert (*Your child "Test User" is near an asthma trigger!*) via their PDA connected to APEX.

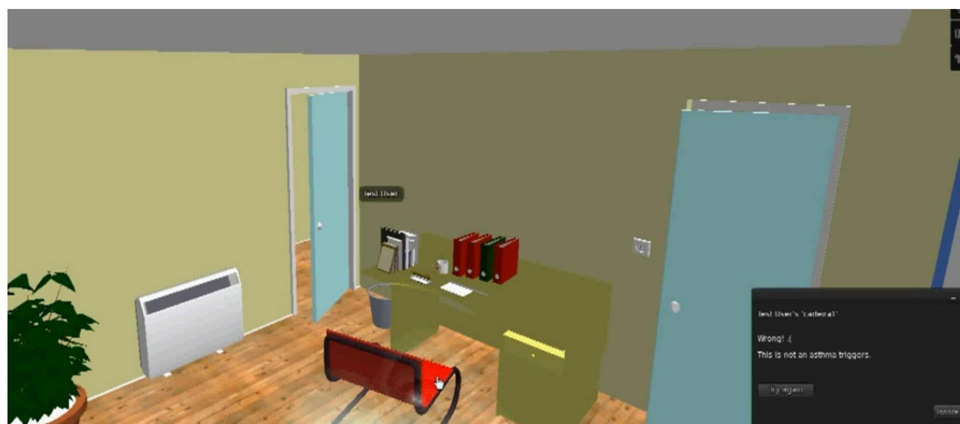

Figure 7.14: Aware Home alert system user experience

The multi-layer approach provides obvious benefits in terms of flexibility of prototyping. As illustrated in this example, depending on the resources available, different user experiences are elicited with various versions of the prototype from the more abstract (PDA as a popup window) to the more immersive (physical PDA). This approach is flexible in the sense that it focuses on providing user experience whatever resources are available.

Figure 7.15: Asthma trigger parent's alert via their PDA

7.3 Serious Games Development

The APEX framework was also used to develop applications other than ubicomp systems’ prototypes, in particular serious games. This section shows how APEX was successfully used in the creation of an educational game.

To illustrate the applicability of APEX in the development of serious games an example based on the aware home is used. The aim of the proposed game is to improve health education of child asthma sufferers and their parents to improve the quality of their lives. This game is played out in a simulation of the aware home, where APEX capabilities are used to create a game.

Hong et al. proposed the creation of an interactive adventure game (*TriggerHunter*) to improve the self-management skills of asthmatic children through an enjoyable experience [95]. The goal of the proposed game is to educate children and their parents about identifying asthma triggers and to learn how to control them. The original game idea is to use mobile phones equipped with a “camera” that identifies asthma triggers and provides information on their screen about the triggers with right and wrong possible options (see Figure 7.16). The proposed game suggests the use of augmented reality and marker tags to identify real world triggers to support being played at home. For more details about the proposed game see [95].

To avoid a costly development and deployment of the game without reducing its purpose an alternative solution was developed with APEX.

The APEX-based game uses a 3D virtual representation of the asthma children sufferer's home instead of using their physical home. The aware home was used for this purpose. Then the goal of the game is to identify asthma triggers present in the home and select from the provided options the appropriate action to deal with them (see Figure 7.17). The user controls an avatar that moves around the house and must find asthma triggers by clicking on objects that compose the environment. Depending on user errors (discovering a trigger but managing it incorrectly) faults are attributed. Each time an asthma trigger is identified relevant information about it is presented providing a more enjoyable way of learning about them. The game finishes when all triggers are identified. The game is easily configurable, so new asthma triggers and corresponding questions can be inserted into the environment using the viewer. As previously stated asthma triggers occur when relevant conditions in the environment are met [97]. By being aware of all existing triggers, individuals know how to act when dealing with their own asthma triggers.

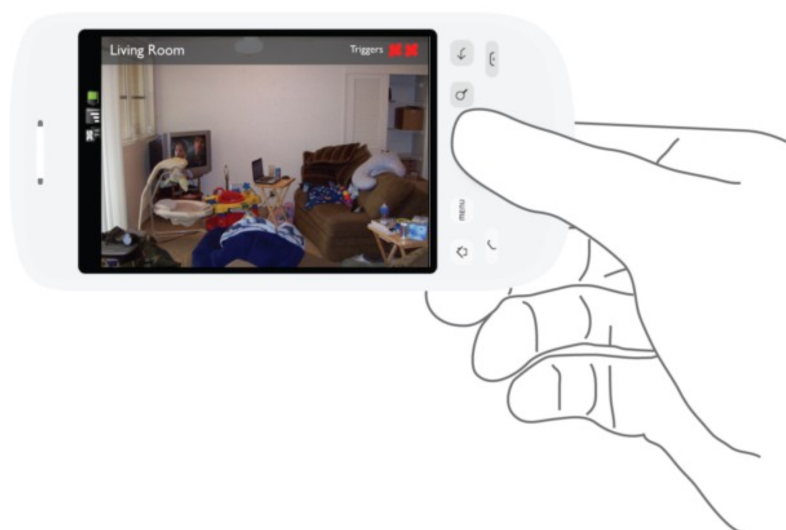

Figure 7.16: *TriggerHunter* game snapshot (adapted from [95])

The game was easily created on top of the virtual environment using APEX by adding to the *CPN base model* a module responsible for the logic of the game (see Figure 7.18). This module receives information from the environment when the child selects an object as being an asthma trigger. If the selected object is an asthma trigger the module decrements the number of remaining triggers to discover and send this information to the virtual environment to

show it to the user. The APEX *communication/execution* component (see Section 3.1.1) is responsible for managing this exchange of information. When an object that is selected is not an asthma trigger the module counts it as a fault. At the end, when all asthma triggers are discovered the number of faults is displayed to the user indicating their performance.

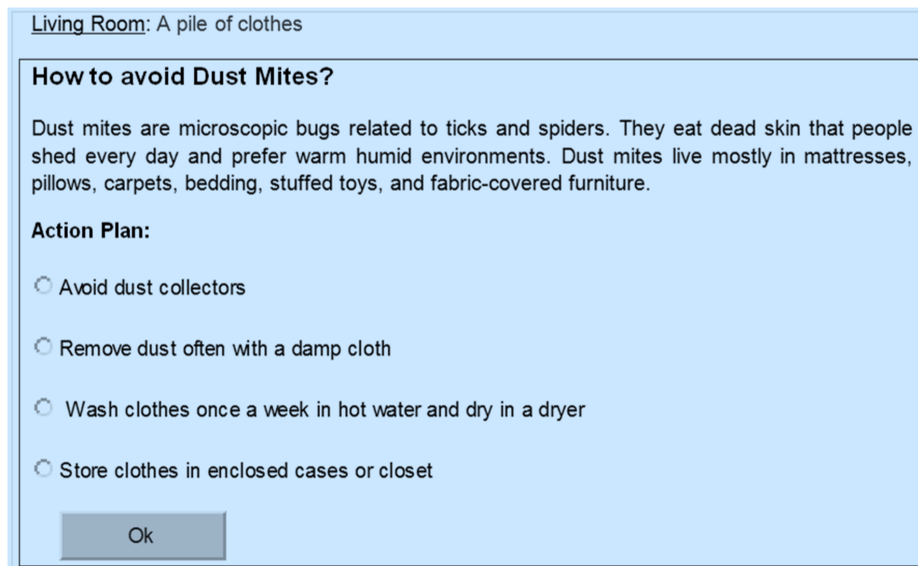

Figure 7.17: *TriggerHunter* game asthma trigger management

Questions and alternative responses are displayed when the user selects an asthma trigger. These are associated with objects using LSL scripts. Figure 7.19 shows a snapshot of the *TriggerHunter* APEX-based game where an avatar selects an asthma trigger and the associated question is displayed. A video showing the usage of the game is accessible at the APEX website[48].

The example demonstrates that APEX seems a promising approach for the rapid development of serious games.

7.4 Conclusions

This chapter introduced a method of evaluating ubicomp environments through exhaustive analysis, applying and adapting heuristics chosen from other areas of HCI. Ubicomp envi-

[48] APEX website: http://ivy.di.uminho.pt/apex (last accessed: 9 February 2012)

ronments pose new challenges when compared with traditional interactive systems. The approach enables the successful exhaustive analysis of ubicomp environments through property patterns. These patterns were instantiated with different values in the context of different ubicomp environments, leading to the identification of procedures to verify a set of properties. The proposed property algorithm skeletons aim to help developers reuse predefined queries over the reachability graph facilitating the verification of properties using APEX.

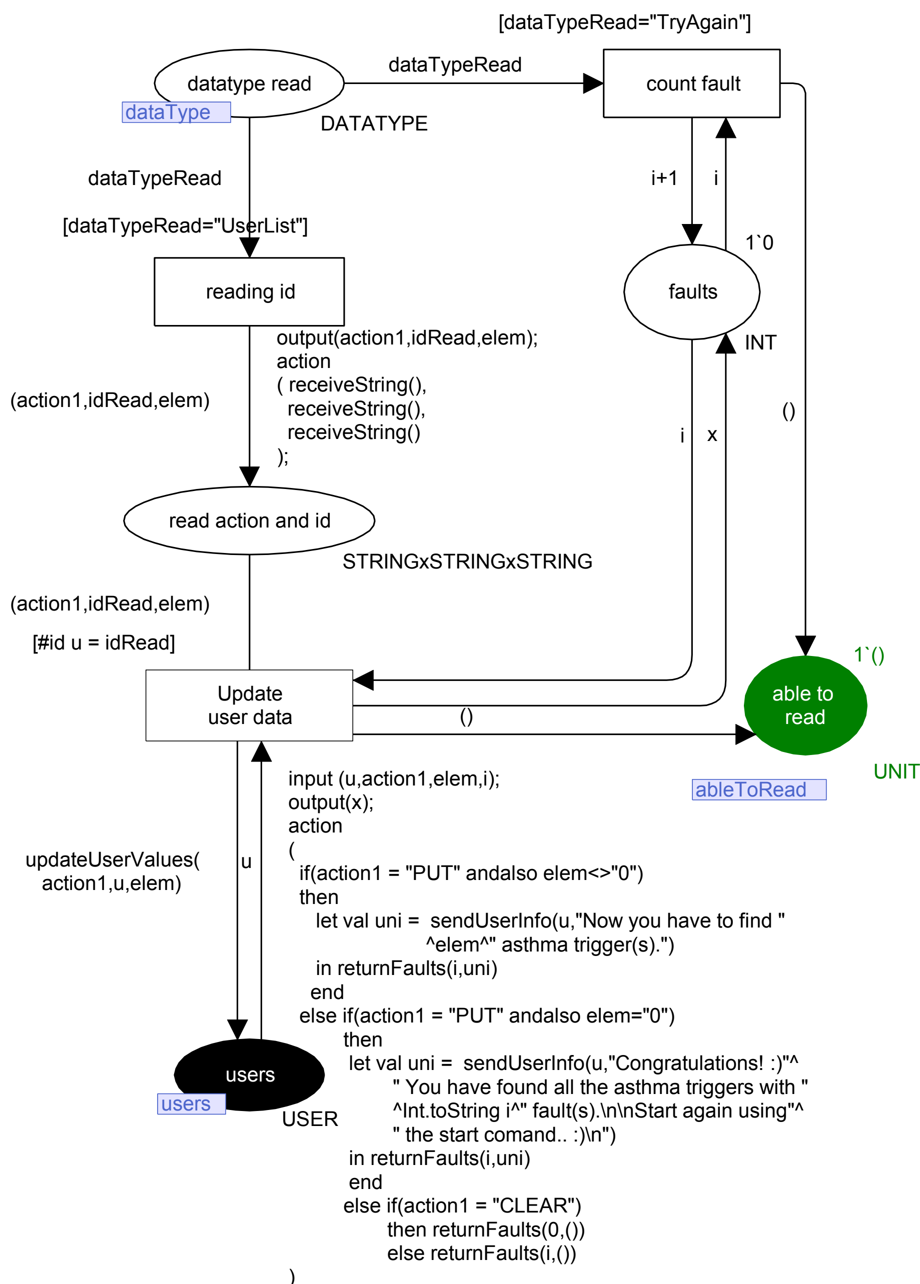

Figure 7.18: Game module

Figure 7.19: Trigger Hunter game using APEX - asthma trigger discovered

APEX, by using CPN, provides a way to analyse exhaustively and formally every portion of the system behaviour for selected scenarios and to verify properties on it. The verification of properties with APEX is limited to the information present in the models. Functional properties are verified based on the information explicitly modelled (e.g. device behaviour specification). Non-functional properties are verified based on the information present in the environment but not necessarily explicitly modelled in the modelling layer. For example the position of dynamic objects in the environment is automatically loaded into the modelling layer and does not involve explicit modelling. These properties were presented as an important feature because they enable the verification of additional aspects of ubicomp environments. For example to consider aspects such as line of sight formally the models would need to be enriched with architectural information, however modelling physical characteristics of the space was not the aim here (it would overly complicate the modelling). Aspects such as line of sight can be dealt with in the 3D simulation.

The APEX multi layer approach complements this exhaustive analysis by providing support for different types of analysis (e.g. user experience provided by the 3D simulation and/or physical layer). Introducing physical devices to the prototype increases the realism of the experience. APEX provides support for physical devices connected through Bluetooth. Their usage improves realism because they are part of the final physical environment being prototyped. Users are able to detect usability issues and design implications by this means. This could be more difficult to detect using virtual physical devices. The analysis based on 3D simulation also provides results such as usability, user experience and feedback that are difficult to obtain using formal analysis based on the SS tool. The focus of concern here is to give a high level of realism to the experience provided.

APEX has also been shown as a promising approach for the rapid development of Serious Games. A game for educational purposes developed with APEX was used to indicate this promise. Further work will evaluate the game with children.

User experience is explored through multi level analysis and an iterative cycle of prototyping (see Figure 7.20) provided by APEX as demonstrated in these examples. The tool therefore responds to the research questions of Section 1.3. It provides reasoning, analysis and experience support through a cycle of development (design, test and analysis).

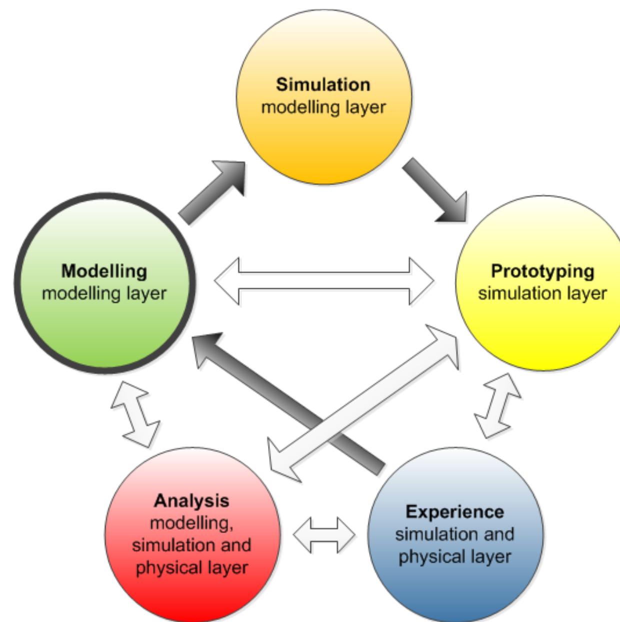

Figure 7.20: APEX - iterative cycle of prototyping (design, test and analysis)

Chapter 8

Evaluation of the Prototyping Approach

To assess the APEX tool and the proposed prototyping approach an evaluation using an example was performed. The evaluation accomplished and presented in this chapter is a preliminary experiment. A more developed experiment will occur later in the broader project in which this thesis work was set. The evaluation made was based on the use of the tool in the context of different alternative solutions for the development of a ubicomp environment. The main goals of the evaluation were to observe how acceptable the tool was to software engineers, how the system is likely to be used, and to identify the aspects that should be improved.

8.1 Example

The goal of the proposed ubiquitous environment is to help elderly persons to go to the bathroom at night (see Section 6.2.3). This is achieved by using presence sensors and lights. The focus of concern was using APEX for the comparison of different solutions to solve the problem of creating a ubiquitous environment addressing the above goal. The developers who participated in the evaluation were asked to produce three alternative solutions for the stated problem. The solutions varied in terms of number, features and position of presence sensors used and the necessary conditions to turn on and off the lights. The proposed exercises can be consulted in Appendix D.

8.2 Relevant User Study Techniques

There are several user study techniques in the literature aimed at evaluating systems. Consolvo et al. [98] proposed a list of relevant techniques to the evaluation of ubiquitous environments. More information about user study techniques can be found in [99, 100]. The particular techniques proposed by Consolvo et al. are listed below.

Intensive Interviewing

This evaluation technique consists in interviewing users about the tasks they have accomplished using open-ended questions. The structure of the questions can vary in content and order from on user to another. The advantages of this technique are that it is:

i. not necessary to use it during the usage of the system by users;
ii. not expensive when compared with other user study techniques.

A main disadvantage is that sometimes users forget some details of their experience of using the system and consequently important aspects may not be mentioned.

Contextual Field Research

This technique is used to discover how users think and act. Data is obtained by interacting with and observing users as part of their normal activity i.e., there is no need for a special environment or conditions to perform the evaluation. Data are typically collected by video, photography, note-taking and/or audio.

This technique has some disadvantages because users know that they are being observed and can consequently change their usual behaviour. Additionally, the technique is more expensive than *Intensive Interviewing*. Despite these drawbacks, evaluators do not have to rely on the memory of users about the tasks they have accomplished. In our case, a further drawback is that APEX provides a novel prototyping approach that developers will not have used as part of their normal activity.

Usability Testing

This technique consists in observing users while they are performing a planned task relating to the system that is being evaluated. Data are collected by observation (e.g. video recording). Two disadvantages are that users know that they are being observed and the scenarios being analysed are planned which means that results in practice could be different. Consolvo et al. argue that it is not the best solution for the evaluation of ubiquitous environments [98].

Lag Sequential Analysis

This technique is similar to *Context Field Research* but with an important difference that data are captured just by observing users without interacting with them. The observation can be made by video recording or note-taking. As with other techniques users know that they are observed.

In their research, Consolvo et al. [98] used this technique for the evaluation of a ubicomp environment and obtained adequate results. However they also stated that the technique is quite expensive mainly because of the expense of video coding.

Questionnaire

Although not mentioned in the original list proposed by Consolvo et al. [98], this technique is relevant and worth mentioning. This technique, like Intensive Interviewing, is applied after the system has been used. It collects users' opinions by asking them to complete a questionnaire. The questionnaire is composed of open-ended questions and questions with a finite set of options. This technique differs from Intensive Interviewing in that there is no interaction between the interviewer and the users.

In our evaluation we have selected a mixture of some of the above techniques to reduce costs and to avoid reliance on users' memory. A questionnaire approach combined with observation (e.g. note-taking) was used while users were performing the planned task (*Usability Testing*).

8.3 Process and Questionnaire

Twenty seven post-graduate software engineers at the University of Minho participated in the evaluation. The evaluation process involved:

1. introducing subjects to Coloured Petri nets and APEX;
2. solving a proposed problem using APEX (see Appendix D);
3. producing a prototype based on a provided virtual environment;
4. completing a questionnaire (see Appendix C).

The participants were monitored by taking notes. The aim was to aid understanding of how the APEX framework was used to provide a prototype. The development was interpreted and marked as being divided into four phases:

1. CPN interpretation;
2. CPN development;
3. virtual environment configuration;
4. prototype examination.

The observation of these phases provided an overall understanding of where participants spent their time. The results were inconclusive in providing insight into whether use of the tool followed any clear pattern. However time spent was as follows:

- CPN modelling (39.9%, $\sigma = 9.5$);
- Virtual environment configuration (30.2%, $\sigma = 9.2$);
- Prototype examination (15.5%, $\sigma = 6.7$);
- CPN interpretation (14.4%, $\sigma = 4.8$).

Because participants were not familiar with the virtual environment configuration this was significantly time consuming. Unsurprisingly, as participants became more familiar with APEX activities they took less time to complete the solutions. To give an idea of the amount of time used in developing prototypes with APEX the following time estimations are provided. Time needed to:

- extend the CPN *base model* for the stated problem (third solution, relatively low complexity level - 2 presence sensors, 1 time sensor and 5 dynamic objects with two possible states): about 15 minutes (once software engineers were familiar with APEX) based on the evaluation results;

- construct the 3D environment: estimation of about 1 person day (importing objects using the Project Mesh viewer) based on the time we spent constructing it;
- tie the two together: 0 minutes (this is done automatically).

The second part of the evaluation involved the questionnaire (Appendix C). The questionnaire was divided into six parts:

- participant characterization;
- usefulness;
- ease of use of the framework;
- ease of learning of the framework;
- user satisfaction.

Subjects were asked to respond with values from -3 (strong disagree) to +3 (strong agree). A final part of the questionnaire enabled the participant to make comments and included open questions. The questionnaire was based on the USE questionnaire[49] which has been used successfully by many companies, and as part of several dissertation projects to evaluate applications.

8.4 Results

The resulting questionnaire results (see APEX website[50]) indicated that the system is around the average in terms of ease of use (-0.37, $\sigma = 1.28$) and the related notion of user satisfaction (-0.11, $\sigma = 1.22$). As mentioned by participants on the open questions and comments, this can be partially due to difficulty using the CPN Tools interface that differs from the usual non-moded style of interface. Another reported reason is that APEX does not provide a general undo facility and provides little support to prevent users making mistakes. This clearly led to frustration in some cases. On the other hand the tool was considered to be relatively easy to learn (+0.63, $\sigma = 1.35$) and to provide useful features (+1.15, $\sigma = 0.97$). While APEX was considered to provide results that met the participants' goals it can clearly be improved.

---

[49] USE questionnaire: http://www.stcsig.org/usability/newsletter/0110_measuring_with_use.html (last accessed: 15 November 2011)

[50] APEX website: http://ivy.di.uminho.pt/apex (last accessed: 19 March 2012)

The results enabled us to better identify aspects of APEX that should be improved as well as providing information about how the system is likely to be used by computer science engineers. These improvements, particularly in terms of ease of use, are planned as future work. The results indicated, besides some usability improvements, that the tool is feasible and appropriate for usage by developers.

8.5 Conclusions

The evaluation results we were aiming at through this preliminary exploration were broadly satisfied. They pointed out some directions for work and revealed reasonable acceptance by software engineers. However this is a very limited evaluation.

It was observed than the time needed to extend the model for the desired problem was very acceptable for a software engineer familiar with APEX (about 15 minutes for the third solution). On the basis of the experiment software engineers became quickly familiar with APEX. In this case familiarity resulted from a training session of four hours.

In this evaluation the virtual environment was provided. However the most time consuming task is related to its creation. In any case, the time needed to create the prototype is low when compared with the one necessary to develop a physical working ubicomp environment (including the costs of materials) which once developed have very high costs of redesign.

Another evaluation could be performed to look into what people can build using APEX. The threshold and ceiling of the framework (how easy is it for people to start using it, and how complex the results they achieve with the tool) is worth further investigation.

The literature on virtual reality for purposes such as education, training or medical treatment, contains good indications that ubicomp environments provide a rich enough experience to allow relevant feedback (e.g., see [101] for some interesting papers on the applicability of virtual reality to behavioural sciences). Nevertheless, further evaluation with end users is an ongoing part of the APEX project to determine whether generated APEX simulations are adequate to help understand how users would experience a specific proposed ubiquitous computing environment reflecting the physical features of the space to be built. A first study was performed where the results of a depth of view study in a physical space [102] were replicated in a virtual environment.

Chapter 9

Conclusions

This dissertation presents an approach to the rapid prototyping of ubicomp environments. We argue that the thesis answers the research questions proposed in Section 1.3. A summary of the contributions provided and a discussion of the work developed are presented. Finally, directions for future work are listed.

9.1 Answers to Research Questions

The overarching goal of the thesis is to investigate whether:

The ubicomp environment development process can be made easier thereby reducing costs, providing early experience and automated analysis support.

This goal raises three primary research questions as stated in the introductory chapter. They were addressed throughout this document and are now summarized.

- **Question one:** *can a formal model represent ubicomp environments?* This research has demonstrated that the CPN modelling approach is appropriate to model ubicomp environments. By using the developed models and following the provided guidelines the CPN model has been successfully used in the prototyping of two ubicomp environment examples.
- **Question two:** *can ubicomp environments prototypes address features with the potential to assess user experience without physical deployment?* The APEX approach, being multi layered, can provide realistic experience to users using the *physical* and *simulation* layers. Physical components that compose the final ubicomp environment in its location can be experienced by users as part of the simu-

lation. This can be achieved during the prototyping phase using the *physical layer*. The 3D simulation provided by the *simulation layer* aims to provide a textured simulation of the space being prototyped, providing a realistic user experience. Design and user experience issues of the presented examples where identified using the developed prototypes.

- **Question three:** *can ubicomp environments be analysed in the early stages of development providing evaluation results at different levels?* This research has demonstrated that the use of a formal modelling approach for the prototyping of ubicomp environments (*modelling layer*) enables reasoning, formal analysis and evaluation in the early phases of development. Additionally, the *simulation* and *physical* layers provide support for different evaluations of ubicomp environments being prototyped. It has been shown in the examples that the ubicomp environments can be analysed in the early stages of development. The evaluation can be related to design, experience and usability concepts or to formal and exhaustive concepts.

9.2 Summary of Contributions

This work addressed several issues that contribute to the main findings of the thesis, concerned with the rapid prototyping of ubiquitous computing environments. The APEX tool provides reasoning and analysis support, while at the same time enabling an assessment of user experience and immersion. These capabilities can reduce development costs. The APEX framework is the major result of this thesis. It satisfies the expected requirements (see Section 2.1) providing a distinct advance over the state of the art.

The APEX framework links together many missing benefits, when compared with the existing approaches, namely:

- support for the design of the ubicomp environment and exploring alternatives, with a particular emphasis on how users will experience the designs;
- support for analysis either by simulation (similar to program execution) or by checking properties of the CPN models;

- a multilayered development approach addressing different ubicomp environment features separately;
- support for the whole prototyping cycle;
- multi-user support and collaborative features enabling interaction between users.

Figure 9.1 summarises all aspects that were addressed during APEX's development. Analysis support was considered capable of enabling the verification of properties on the developed prototypes. Evaluation with software engineers pointed out the aspects where APEX should be improved. The simulation and physical devices were addressed to provide experience to users. The modelling provides a way of reasoning and formal/exhaustive analysis support. Immersion provides deeper experience to users. Aspects related to costs, the whole cycle of development and ubiquitous computing were also considered in the development of APEX.

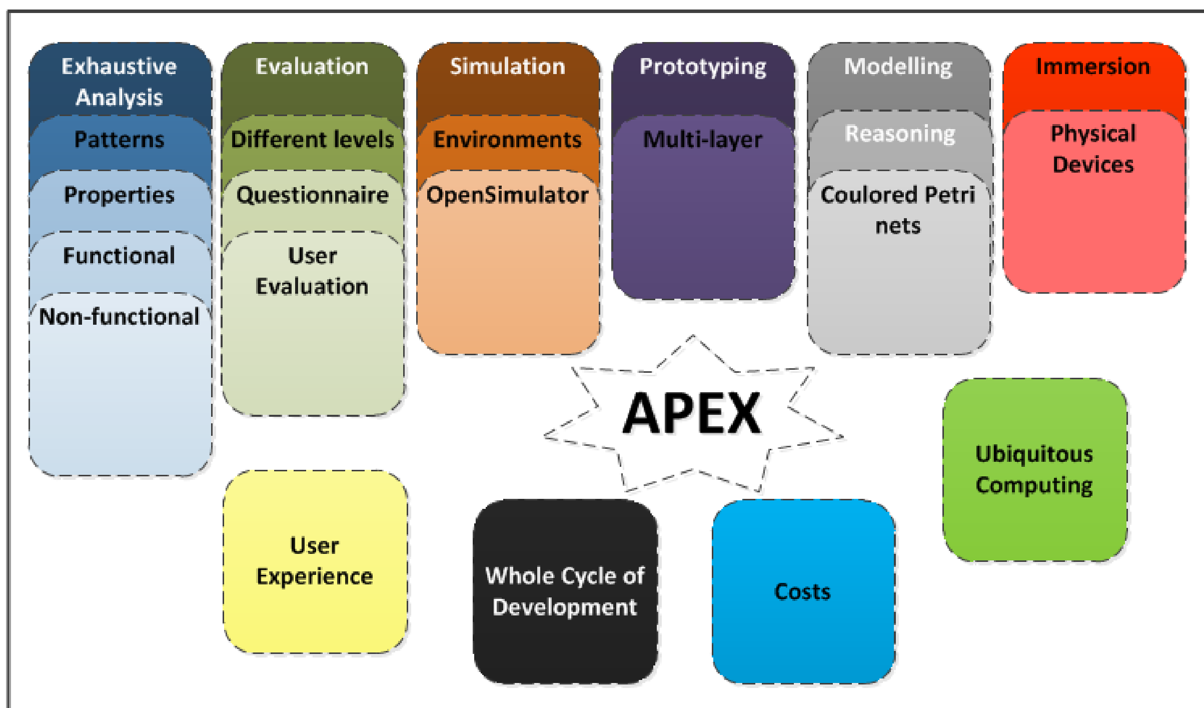

Figure 9.1: APEX fields

The APEX framework proposes a solution to reduce the costs of the development of ubicomp environment. A video showing results achieved using the APEX framework can be consulted at the APEX website[51].

[51] APEX website: http://ivy.di.uminho.pt/apex (last accessed: 19 March 2012)

9.3 Discussion

Since there have been many ubicomp frameworks, architectures, toolkits and tools proposed over recent years (see Chapter 2), it should be clear why the ubicomp/HCI community needs another prototyping framework (see Chapter 2). As stated in Chapter 2, existing approaches already address some aspects provided by the APEX framework. For example VARU provides a multilayered development approach and d.tools provides support for the whole cycle of prototyping. However a framework that provides the list of benefits identified in the previous section in a single framework was missing. APEX provides benefits also provided by other approaches (e.g. multilayered development) and link them with new benefits (e.g. exhaustive analysis support) into an integrated framework.

It is difficult to set up, analyse and experience ubicomp environments in a safe, practical and economical way. APEX is a framework where avatars (programmed or controlled by real users) interact in a 3D ubicomp environment and with physical devices allowing the unexpected to occur. Ubicomp environments can be experienced with APEX in a 3D environment and analysed through exhaustive analysis.

The APEX framework enables the possibility of identifying requirements for a specific ubicomp environment to be built. Additionally, the capacity of the framework to show a 3D simulation to the clients of the ubicomp environment being prototyped instead of, for example, communicating via UML diagrams is a clear advantage. Clients can easily understand the proposed solutions making it easier to indicate directions for improvement. The disadvantage is related to the work associated with the construction of the virtual environment. However, the usage of a viewer supporting mesh objects substantially reduces these costs.

Including a virtual environment in the prototyping process (the *simulation layer*) means prototyping becomes less rapid than when done with models only (the *modelling layer*), due to the time needed to create a virtual environment. However, the fact that it enables movement between layers means that different features of a ubiquitous environment can be explored in a variety of modes. When compared with alternative solutions (e.g. Wizard of Oz or paper prototyping) this approach offers benefits in terms of results such as analysis support and user experience.

It should be noticed that APEX is not focused at the rapid prototyping of all ubicomp application domains but mainly to ubicomp environments characterised as technologically enhanced spaces (e.g. ubicomp airport or hospital).

9.4 Threats and limitations

Ubicomp environments can in principle be simulated without APEX i.e., just using Opensimulator and scripts associated with objects. This is enough to provide user experience to users and identify design issues. However APEX is an integrated framework that support the whole cycle of development, exhaustive analysis, multi-layer development support, physical device connection among other aspects which are much more than just providing user experience. The availability of all these characteristics in only one framework is a benefit in the sense that there no need for other tools for the prototyping of ubicomp environments.

Another remark relates to CPN model simulation. As stated the execution of the model within CPN Tools (similar to a program execution) drives the environment. The delay in milliseconds of each transition execution is configurable however never lower than 1 millisecond. Consequently, possible performance problems can appear when for instance too much information is transmitted from the simulation to CPN Tools (e.g. many users arriving close to a presence sensor at same time). During experiments the information sent from Opensimulator (e.g. user detection by a presence sensor) were so low that it was not an issue. Nevertheless, the assumption that CPN Tools run fast enough to keep up with input from Opensimulator could be a dangerous assumption to make. As future work the number of messages sent from Opensimulator could be optimized (reduced) eliminating possible performance problems (environment' reaction delay to user actions or context changes).

9.5 Future Work

Some future work has already been pointed out in this thesis. A summary of future directions are listed below:

- Properties are verifiable within APEX, however improvement of the APEXi tool to verify properties, without it being necessary to write queries over the model at a

low level of abstraction, will represent an important improvement. The idea would be to specify the queries at a higher level of abstraction following the concepts used by the IVY tool [88];

- The means of interaction with APEX prototypes could provide a even more realistic experience to end users through the integration of more digital devices (e.g. touch screens, interactive whiteboards and digital gloves). Further development of the framework should also focus on the connection of isolated sensors that are not integrated into mobile devices, and support the modelling of a richer set of interactions of programmed avatars;
- Experience results are obtained with APEX mainly based on the virtual experience provided. However, it is important to check if the results acquired reflect the experience obtained while experiencing the final environment on location. The application of the APEX framework to prototyping an existing physical ubicomp environment is a topic to consider that will indicate the eventual limits of the approach and point areas for improvement. For example, the extension of APEX to the internet of things [103] in the sense that it will enable the prototyping and analysis of physical ubicomp environments composed of elements which are connected to the internet can be an adequate direction to follow. Additionally, more evaluations of the framework not only with developers but also with end users will emphasize the assessment of APEX for developers and end users usage.

Bibliography

[1] J. L. Silva, J. C. Campos, and M. D. Harrison, "An infrastructure for experience centered agile prototyping of ambient intelligence," in *Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems*, 2009, pp. 79–84.

[2] J. L. Silva, Ó. Ribeiro, J. Fernandes, J. Campos, and M. Harrison, "The APEX framework: prototyping of ubiquitous environments based on Petri nets," in *Human-Centred Software Engineering. Lecture Notes in Computer Science. Springer*, 2010, vol. 6409, pp. 6–21.

[3] J. L. Silva, Ó. R. Ribeiro, J. M. Fernandes, J. C. Campos, and M. D. Harrison, "Prototipagem rápida de ambientes ubíquos," in *4a. Conferência Nacional em Interacção Humano-Computador (Interacção 2010)*, 2010, pp. 121-128.

[4] M. Weiser, "The computer for the 21st century," *Scientific American*, vol. 265, no. 3, pp. 94-104, 1991.

[5] T. Jokela, N. Iivari, and J. Matero, "The standard of user-centered design and the standard definition of usability: analyzing ISO 13407 against ISO 9241-11," *the Latin American conference*, pp. 53-60, 2003.

[6] G. D. Abowd, E. D. Mynatt, and T. Rodden, "The human experience," *Pervasive Computing, IEEE*, vol. 1, no. 1, pp. 48–57, Jan. 2002.

[7] K. Harrison, M., Campos, J., Doherty, G., Loer, "Connecting rigorous system analysis to experience centred design," in *Maturing Usability: Quality in Software, Interaction and Value. Springer-Verlag*, 2008, pp. 56-74.

[8] D. Garlan, D. P. Siewiorek, A. Smailagic, and P. Steenkiste, "Project aura: Toward distraction-free pervasive computing," *Pervasive Computing, IEEE*, vol. 1, no. 2, pp. 22–31, 2002.

[9] S. H. and A. S. N. Davies, J. Landay, "Special Issue of IEEE Pervasive Computing on Rapid Prototyping for Ubiquitous Computing," 2005, vol. 4, no. 4.

[10] J. J. Barton and V. Vijayaraghavan, "UBIWISE, a simulator for ubiquitous computing systems design," *Hewlett-Packard Laboratories Palo Alto, HPL-2003-93*, 2003.

[11] H. Nishikawa et al., "UbiREAL: Realistic smartspace simulator for systematic testing," *Lecture Notes in Computer Science*, vol. 4206/2006, pp. 459–476, 2006.

[12] B. Hartmann et al., "Reflective physical prototyping through integrated design, test, and analysis," in *Proceedings of the 19th annual ACM symposium on User interface software and technology*, 2006, pp. 299–308.

[13] Y. Li and J. I. Hong, “Topiary: a tool for prototyping location-enhanced applications,” *Proceedings of the 17th annual ACM*, vol. 6, no. 2, pp. 217–226, 2004.

[14] D. Salber, A. K. Dey, and G. D. Abowd, “The context toolkit: aiding the development of context-enabled applications,” in *Proceedings of the SIGCHI conference on Human factors in computing systems: the CHI is the limit*, 1999, pp. 434–441.

[15] G. D. Abowd et al., “Prototypes and paratypes: Designing mobile and ubiquitous computing applications,” *Pervasive Computing, IEEE*, vol. 4, no. 4, pp. 67–73, 2005.

[16] D. Kang, K. Kang, H. Lee, and E.-jung Ko, “A Systematic Design Tool of Context Aware System for Ubiquitous Healthcare Service in a Smart Home,” in *Future Generation Communication and Networking*, 2007, vol. 2, p. 49--54.

[17] S. Nazari and A. Klar, “3DSim: Rapid Prototyping Ambient Intelligence,” in *SOc-EUSAI conference*, 2005, pp. 303–307.

[18] T. Disz and M. E. Papka, “UbiWorld: an environment integrating virtual reality, supercomputing, and design,” *Computing Workshop*, pp. 46–57, 1997.

[19] E. O’Neill, D. Lewis, and O. Conlan, “A simulation-based approach to highly iterative prototyping of ubiquitous computing systems,” in *2nd International Conference on Simulation Tools and Techniques*, 2009, p. 56--66.

[20] E. O’Neill, “Master Thesis: TATUS a Ubiquitous Computing Simulator,” University of Dublin, 2004.

[21] S. Irawati, S. Ahn, J. Kim, and H. Ko, “Varu framework: Enabling rapid prototyping of VR, AR and ubiquitous applications,” in *Virtual Reality Conference, 2008. VR’08. IEEE*, 2008, pp. 201–208.

[22] L. Vanacken, J. De Boeck, C. Raymaekers, and K. Coninx, “Designing context-aware multimodal virtual environments,” in *Proceedings of the 10th international conference on Multimodal interfaces*, 2008, pp. 129-136.

[23] Y. Li and J. A. Landay, “Into the wild: low-cost ubicomp prototype testing,” *Computer*, vol. 41, no. 6, pp. 94–97, 2008.

[24] T. Sohn, “iCAP: an informal tool for interactive prototyping of context-aware applications,” *CHI 03 extended abstracts on Human factors in computing systems*, pp. 974-975, 2003.

[25] S. Carter, J. Mankoff, and J. Heer, “Momento: support for situated ubicomp experimentation,” in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2007, pp. 125–134.

[26] P. Singh, H. Ha, P. Olivier, C. Kray, and Z. Kuang, "Rapid prototyping and evaluation of intelligent environments using immersive video," in *Proceedings of MODIE workshop at Mobile HCI'06*, 2006.

[27] "Simulation." [Online]. Available: http://en.wikipedia.org/wiki/Computer_simulation. [Accessed: 15-Nov-2011].

[28] E. Dubois, P. Gray, and L. Nigay, "ASUR ++ : A Design Notation for Mobile Mixed Systems," in *Proceedings of the 4th International Symposium on Mobile Human-Computer Interaction*, 2002, pp. 123-139.

[29] E. Dubois and P. Gray, "A design-oriented information-flow refinement of the ASUR interaction model," *Engineering Interactive Systems*, pp. 465–482, 2008.

[30] R. Wieting, "Hybrid high-level nets," in *Proceedings of the 28th conference on Winter simulation*, 1996, pp. 848–855.

[31] M. Massink, D. Duke, and S. Smith, "Towards hybrid interface specification for virtual environments," in *Design, Specification and Verification of Interactive Systems*, 1999, vol. 99, pp. 30–51.

[32] D. Navarre et al., "A formal description of multimodal interaction techniques for immersive virtual reality applications," in *Proceedings of the 2005 IFIP TC13 international conference on Human-Computer Interaction*, 2005, pp. 170–183.

[33] R. Bastide, D. Navarre, P. Palanque, A. Schyn, P. Dragicevic, and U. Toulouse, "A Model-Based Approach for Real-Time Embedded Multimodal Systems in Military Aircrafts," in *Proceedings of the 6th international conference on Multimodal interfaces*, 2004, pp. 243-250.

[34] L. M. Jensen, Kurt, Kristensen, *Coloured Petri Nets Modelling and Validation of Concurrent Systems*. Springer, 2009, p. 384.

[35] C. Hoare, *Communicating sequential processes*. Prentice Hall International, 2004, p. 260.

[36] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of computer programming*, vol. 8, pp. 231-274, 1987.

[37] K. C. Erwin Cuppens, Chris Raymaekers, "VRIXML: A User Interface Description Language for Virtual Environments," in *Developing User Interfaces with XML: Advances on User Interface Description Languages*, 2004, pp. 111-117.

[38] H. Jed and W. Josie, *The VRML 2.0 handbook: building moving worlds on the web*. Addison Wesley Longman Publishing Co., Inc., 1996, p. 412.

[39] O. De Troyer, F. Kleinermann, B. Pellens, and W. Bille, “Conceptual modeling for virtual reality,” in *Tutorials, posters, panels and industrial contributions at the 26th international conference on Conceptual modeling - Volume 83*, 2007, vol. 83, pp. 3–18.

[40] E. Dubois, L. Nigay, and J. Troccaz, “Consistency in augmented reality systems,” in *Proceedings of the 8th IFIP International Conference on Engineering for Human-Computer Interaction*, 2001, pp. 111-122.

[41] E. Dubois, L. Nigay, J. Troccaz, O. Chavanon, and L. Carrat, “Classification Space for Augmented Surgery. an Augmented Reality Case Study,” in *Human-computer interaction. IFIP TC. 13 International Conference on Human-Computer*, 1999, vol. 1, p. 353--359.

[42] S. Smith, “Virtual environments as hybrid systems,” in *Eurographics UK 17th Annual Conference*, 1999, p. 113--128.

[43] J. Willans, “PhD Thesis. Integrating behavioural design into the virtual environment development process,” University of York, 2002.

[44] K. Jensen and S. Christensen, *CPN Tools State Space Manual*. University of Aarhus, 2006, pp. 1-49.

[45] A. Ratzer et al., “CPN tools for editing, simulating, and analysing coloured Petri nets,” *Applications and Theory of Petri Nets*, pp. 450–462, 2003.

[46] B. V. Schooten, O. Donk, and J. Zwiers, “Modelling interaction in virtual environments using process algebra,” in *WLT15: Interactions in Virtual Worlds*, 1999, p. 195--212.

[47] L. Briand and C. Williams, Eds., “Model Driven Engineering Languages and Systems,” in *8th International Conference MoDELS*, 2005.

[48] M. Crane and J. Dingel, “UML vs. classical vs. Rhapsody statecharts: Not all models are created equal,” *Model Driven Engineering Languages and Systems*, vol. 1, pp. 97–112, 2005.

[49] R. Eshuis, “Statecharting petri nets.” Technische Universiteit Eindhoven, pp. 1-37, 2005.

[50] J. Kienzle, A. Denault, and H. Vangheluwe, “Model-based design of computer-controlled game character behavior,” in *MoDELS*, 2007, pp. 650–665.

[51] D. Harel and M. Politi, *Modeling reactive systems with statecharts: the statemate approach*. McGraw-Hill, Inc., 1998.

[52] “The Mathworks. Stateflow and stateflow coder users guide,” 2005. [Online]. Available: http://www.mathworks.com. [Accessed: 23-Feb-2012].

[53] A. Egyed and D. Wile, “Statechart simulator for modeling architectural dynamics,” in *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, 2001, pp. 87–96.

[54] K. Fuhrmann and J. Hiemer, “Formal verification of STATEMATE-statecharts,” in *Environment*, 2001.

[55] T. BASTEN, “PhD Thesis. In Terms of Nets System Design with Petri Nets and process algebra,” Eindhoven University of Technology, 1998.

[56] R. H. and D. M. Robin Milner, Mads Tofte, *The Definition of Standard ML, Revised Edition*. The MIT Press, 1997, p. 128.

[57] “Very Brief Introduction to Coloured Petri Nets.” [Online]. Available: http://daimi.au.dk/CPnets/proxy.php?url=/CPnets/intro/verybrief. [Accessed: 28-Jan-2012].

[58] K. Jensen, L. M. Kristensen, and L. Wells, “Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems,” *International Journal on Software Tools for Technology Transfer*, vol. 9, no. 3-4, pp. 213-254, Mar. 2007.

[59] “Recommended Books and Papers on Coloured Petri Nets.” [Online]. Available: http://www.daimi.au.dk/~kjensen/papers_books/rec_papers_books.html#intro_cpn_papers. [Accessed: 28-Jan-2012].

[60] P. M. Lester and C. M. King, “Analog vs. Digital Instruction and Learning: Teaching Within First and Second Life Environments,” *Journal of Computer-Mediated Communication*, vol. 14, no. 3, pp. 457-483, Apr. 2009.

[61] J. L. McBrien, R. Cheng, and P. Jones, “Virtual spaces: Employing a synchronous online classroom to facilitate student engagement in online learning,” *The International Review of Research in Open and Distance Learning*, vol. 10, no. 3, p. 17, 2009.

[62] “Virtual worlds registration.” [Online]. Available: http://www.slideshare.net/nicmitham/kzero-universe-chart-q3-2010. [Accessed: 15-Nov-2011].

[63] F. Scott, “PhD Thesis: An Investigation into the Ecological Validity of Virtual Reality Measures of Planning and Prospective Memory in Adults with Acquired Brain Injury and Clinical Research Portfolio,” University of Glasgow, 2011.

[64] B. Orland, K. Budthimedhee, and J. Uusitalo, “Considering virtual worlds as representations of landscape realities and as tools for landscape planning,” *Landscape and Urban Planning*, vol. 54, no. 1-4, pp. 139-148, May 2001.

[65] S. D. Freitas, “Serious virtual worlds: a scoping study,” *Prepared for the JISC e-learning programme*, p. 52, 2008.

[66] "OpenSimulator Based Projects." [Online]. Available: http://wiki.jokaydia.com/page/Vws_list. [Accessed: 15-Nov-2011].

[67] "Game Engines." [Online]. Available: http://en.wikipedia.org/wiki/List_of_game_engines. [Accessed: 15-Nov-2011].

[68] J. Nielsen, "Enhancing the explanatory power of usability heuristics," in *Proceedings of the SIGCHI conference on Human factors in computing systems: celebrating interdependence*, 1994, pp. 152–158.

[69] R. Rukšenas, J. Back, P. Curzon, and A. Blandford, "Formal modelling of salience and cognitive load," in *Proc. 2nd Int. Workshop on Formal Methods for Interactive Systems: FMIS*, 2007, pp. 57–75.

[70] S. Kim and S. Kim, "Usability challenges in ubicomp environment," in *The Proceeding of International Ergonomics Association (IEA'03)*, 2003, p. 4.

[71] J. Mankoff, A. Dey, G. Hsieh, and J. Kientz, "Heuristic evaluation of ambient displays," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2003, no. 5, pp. 169-176.

[72] J. Scholtz and S. Consolvo, "Toward a framework for evaluating ubiquitous computing applications," *Pervasive Computing, IEEE*, vol. 3, no. 2, pp. 82–88, 2004.

[73] M. Theofanos and J. Scholtz, "A Framework for Evaluation of Ubicomp Applications," in *First International Workshop on Social Implications of Ubiquitous Computing, CHI*, 2005, pp. 1-5.

[74] J. Scholtz, L. Arnstein, M. Kim, T. Kindberg, and S. Consolvo, "User-Centered Evaluations of Ubicomp Applications," *Intel Corporation*, vol. 10, 2002.

[75] E. Kindler and C. Pále, "3D-visualization of Petri net models: A concept," in *Workshop Algorithmen und Werkzeuge für Petrinetze*, 2003, p. 464--473.

[76] "OpenSimulator compatible viewers." [Online]. Available: http://opensimulator.org/wiki/Connecting. [Accessed: 13-Feb-2012].

[77] "Comms/CPN: A communication infrastructure for external communication with design/CPN," in *Kurt Jensen (Ed.): 3rd Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN'01)*, 2001, pp. 75-90.

[78] M. Westergaard and K. Lassen, "The britney suite animation tool," in *Proceedings of the 27th international conference on Applications and Theory of Petri Nets and Other Models of Concurrency*, 2006, pp. 431–440.

[79] L. Paganelli and F. Paternò, "Tools for remote usability evaluation of Web applications through browser logs and task models.," *Behavior research methods, instru-*

ments, & computers : a journal of the Psychonomic Society, Inc, vol. 35, no. 3, pp. 369-78, Aug. 2003.

[80] "ISO DIS 9241-210:2008 [3] standard." .

[81] E. Law, V. Roto, and M. Hassenzahl, "Understanding, scoping and defining user experience: a survey approach," in *Proceedings of the 27th international conference on Human factors in computing systems*, 2009, pp. 719-728.

[82] R. Moreira, "Master Thesis: Integrating a 3D application server with a CAVE," University of Minho, 2011.

[83] K. Haynes, J. Morie, and E. Chance, "I want my virtual friends to be life size!: adapting Second Life to multi-screen projected environments," in *ACM SIGGRAPH 2010 Posters*, 2010, vol. 2004, p. 60--60.

[84] D. Bowman, "Virtual reality: how much immersion is enough?," *Computer*, vol. 40, no. 7, pp. 36 - 43, 2007.

[85] M. W. Newman et al., "Bringing the Field into the Lab: Supporting Capture and RePlay of Contextual Data for Design," *Human Factors*, pp. 105-108, 2010.

[86] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Proceedings of the 21st international conference on Software engineering*, 1999, pp. 411-420.

[87] J. Campos and M. Harrison, "Systematic analysis of control panel interfaces using formal tools," in *Interactive Systems. Design, Specification, and Verification*, 2008, pp. 72–85.

[88] J. Campos and M. D. Harrison, "Interaction engineering using the IVY tool," in *Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems*, 2009, pp. 35–44.

[89] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 8, no. 2, pp. 244–263, Apr. 1986.

[90] H. Thimbleby and P. Oladimeji, "Social network analysis and interactive device design analysis," in *Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems*, 2009, p. 91--100.

[91] A. Cheng, S. Christensen, and K. H. Mortensen, *Model checking coloured petri nets exploiting strongly connected components*, vol. 6. Aarhus Universitet, Datalogisk Afdeling, 1997, pp. 1-14.

[92] J. Bauchot, F., Clement, J.-Y., Marmigere, G., Picon, "Method and structure for localizing objects using daisy chained RFID tags," U.S. Patent US 2007/0257799 A12007.

[93] E. D. Bateman et al., “Global strategy for asthma management and prevention: GINA executive summary.,” *The European respiratory journal : official journal of the European Society for Clinical Respiratory Physiology*, vol. 31, no. 1, pp. 143-78, Jan. 2008.

[94] M. D. Cabana et al., “Parental management of asthma triggers within a child’s environment.,” *The Journal of allergy and clinical immunology*, vol. 114, no. 2, pp. 352-7, Aug. 2004.

[95] H. Hong, H. Y. Jeong, R. I. Arriaga, and G. D. Abowd, “TriggerHunter: designing an educational game for families with asthmatic children,” in *Proceedings of the 28th of the international conference extended abstracts on Human factors in computing systems*, 2010, pp. 3577–3582.

[96] “Asthma triggers.” [Online]. Available: www.cdc.gov/asthma/triggers.html. [Accessed: 15-Nov-2011].

[97] “Center for Disease Control and Prevention.” [Online]. Available: http://www.cdc.gov/asthma/triggers.html. [Accessed: 04-Jan-2012].

[98] S. Consolvo, L. Arnstein, and B. Franza, “User study techniques in the design and evaluation of a ubicomp environment,” in *Proceedings of the 4th international conference on Ubiquitous Computing*, 2002, p. 73--90.

[99] J. A. T. Hackos and J. Redish, *User and task analysis for interface design*, vol. 31, no. 3. Wiley New York, 1998, pp. 19-20.

[100] J. Rubin, D. Chisnell, and J. Spool, *Handbook Of Usability Testing:How To Plan, Design And Conduct Effective Tests*. Wiley India Pvt. Ltd., 2008, p. 384.

[101] “CyberPsychology & Behavior,” *CyberPsychology & Behavior*, vol. 6, no. 3/4, 2003.

[102] T. Varoudis, S. Dalton, K. Alexiou, and T. Zamenopoulos, “Ambient displays: influencing movement patterns,” in *Proceedings of the 2011 annual conference extended abstracts on Human factors in computing systems*, 2011, p. 1225--1230.

[103] L. Atzori and A. Iera, “The internet of things: A survey,” *Computer Networks*, vol. 54, no. 15, pp. 2787-2805, Oct. 2010.

[104] G. Flaus, Jean-Marie and Ollagnon, “Hybrid flow nets for hybrid processes modelling and control,” *Hybrid and Real-Time Systems*, vol. 1201, pp. 213-227, 1997.

Appendix A: CPN Base Model

This section describes relevant modules of the CPN *Base model*. The full model can be consulted in the Compact Disk attached to this document.

The module of Figure A.0.1 presents the overview of the elements that can compose the ubicomp environment (e.g. users, dynamic objects and sensors) and is responsible to initialize the model execution.

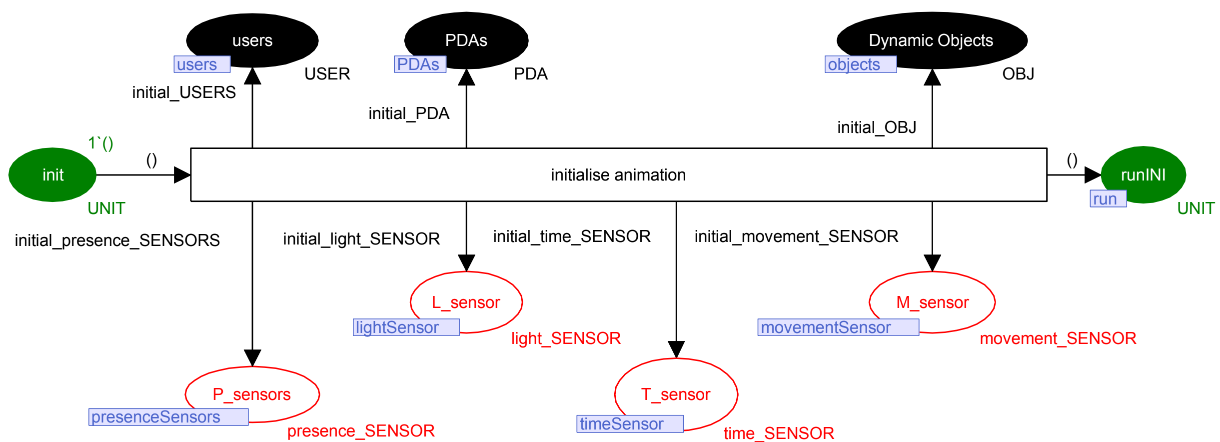

Figure A.0.1: *CPN base* initialization module

The module of Figure A.0.2 is responsible for the reading of information from Opensimulator.

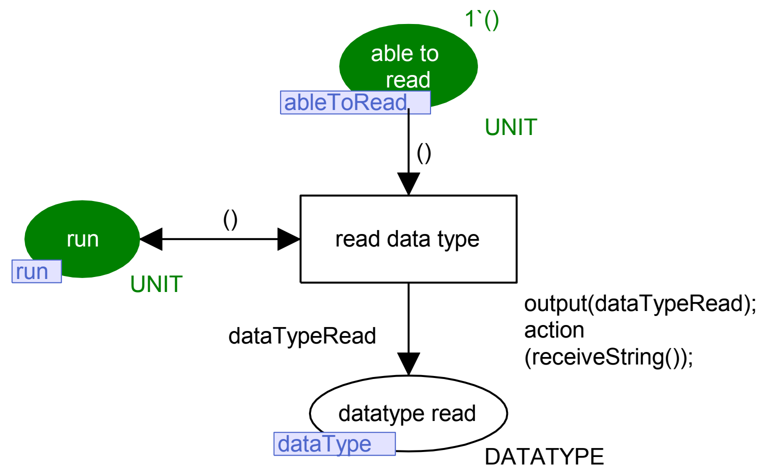

Figure A.0.2: *CPN base* data type reading module

The module of Figure A.0.3 is responsible to update the values of the movement sensors tokens present in the model in agreement with the users movement in the simulation layer.

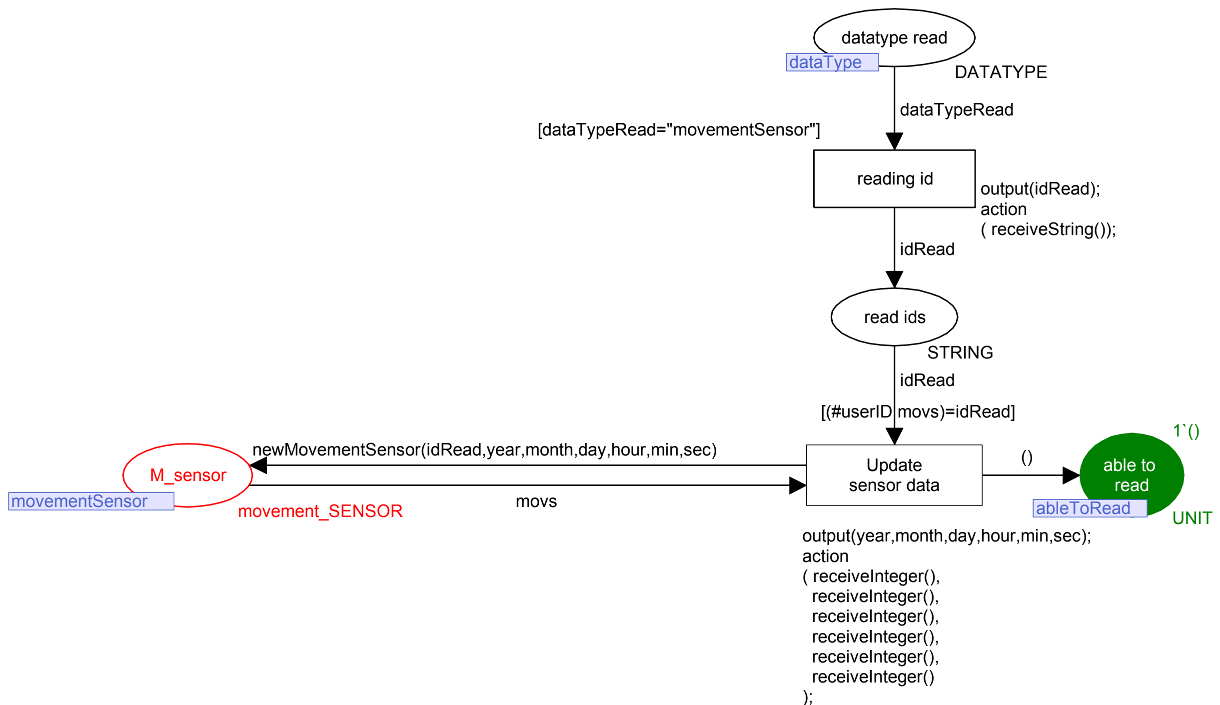

Figure A.0.3: *CPN base* update movement sensors module

The module of Figure A.0.4 is responsible to synchronize the sensing of avatars by the presence sensors in the 3D environment with the tokens in the model that represent them.

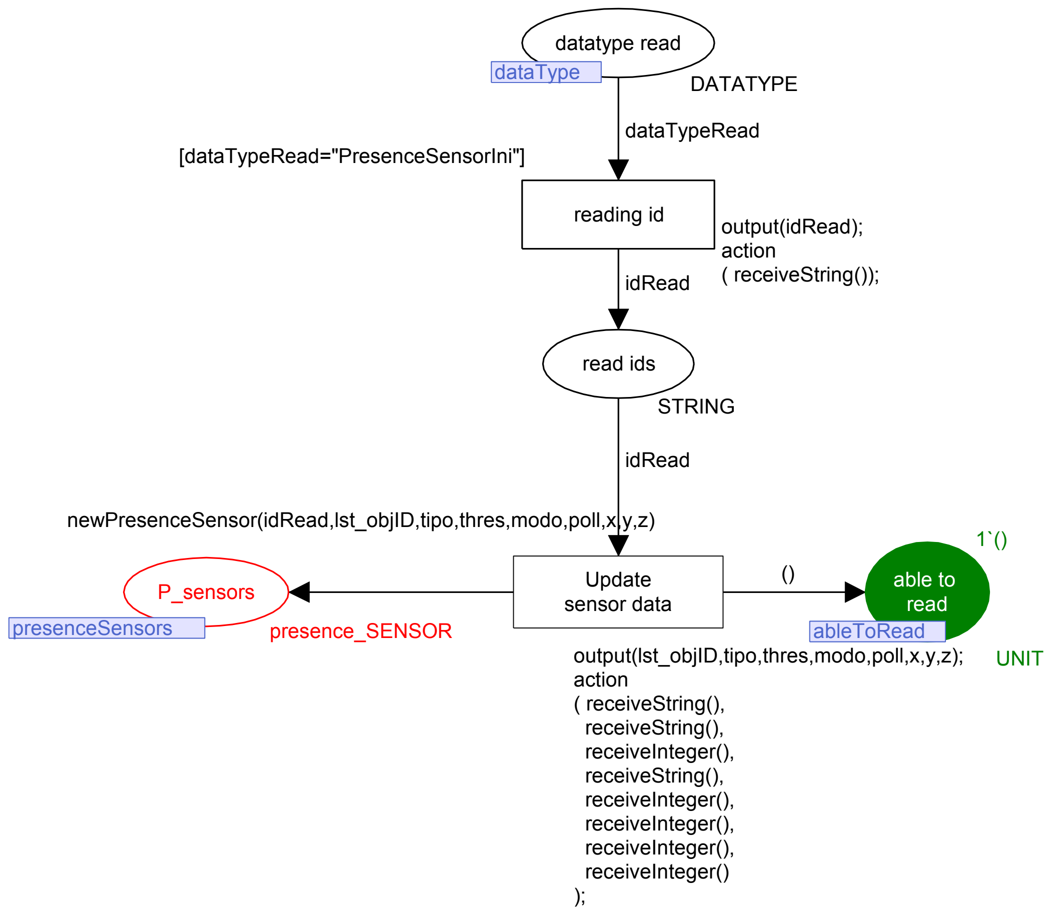

Figure A.0.4: *CPN base* update presence sensors module

Appendix B: CPN Analysis Model

This section describes the main modules of the CPN *Analysis model*. The full model can be consulted in the Compact Disk attached to this document.

The Figure B.0.1 presents the module responsible to update, the tokens of the model that represent dynamic objects, in agreement with the values provided to the *small colour sets* (*ObjIDs*, *ObjFeatures*).

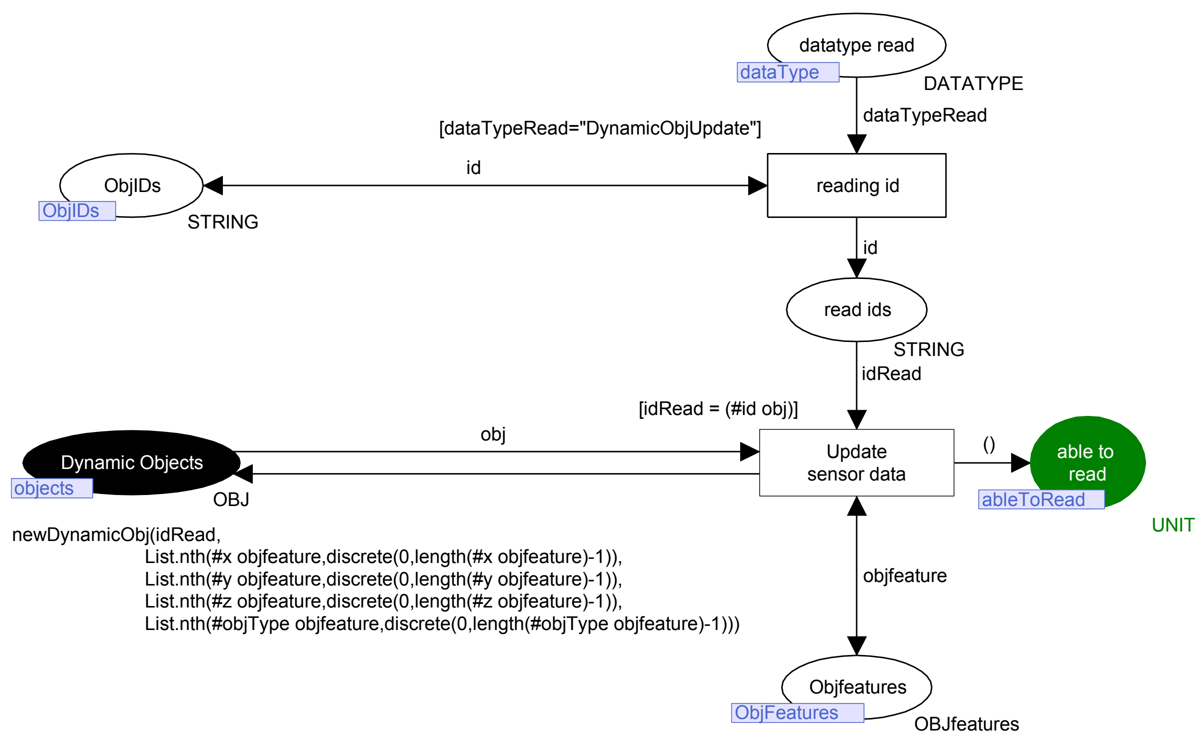

Figure B.0.1: *CPN analysis* dynamic objects update module

The module of Figure B.0.2 represents the module responsible to update the value of the tokens that represent the light sensors of the environment in agreement with the initialized *small colour sets* (*LS_IDs*, *hours*).

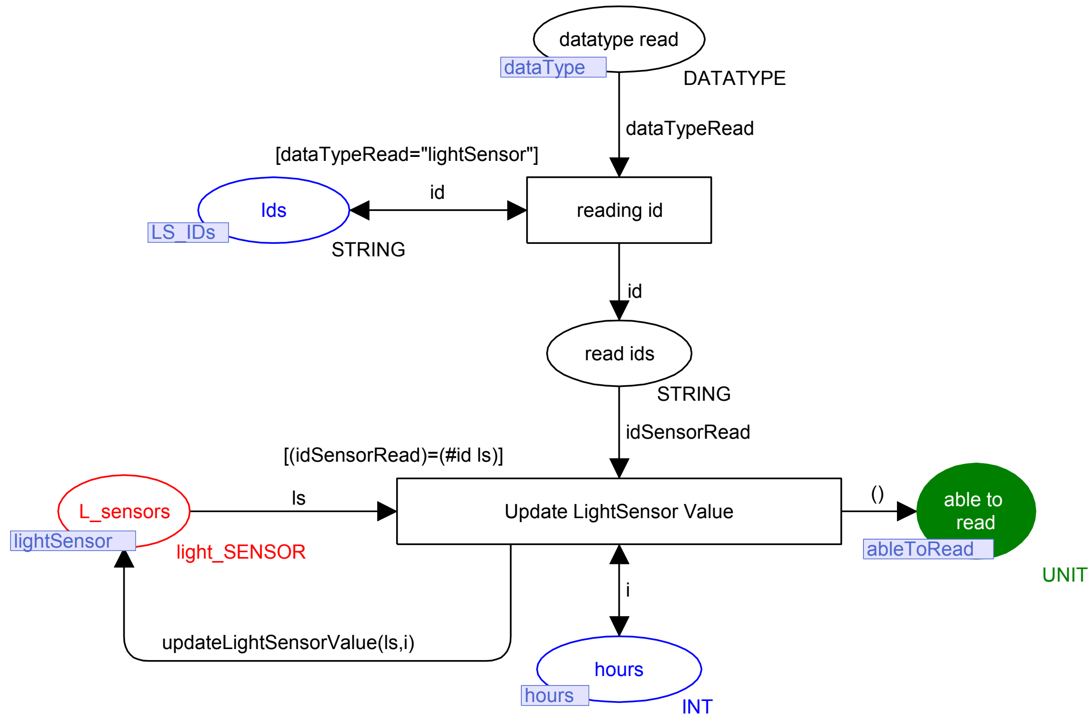

Figure B.0.2: *CPN analysis* light sensors update module

The module of Figure B.0.3 is responsible to update the tokens of the time sensor using the values of the initialized *small colour sets* (*TS_IDs*).

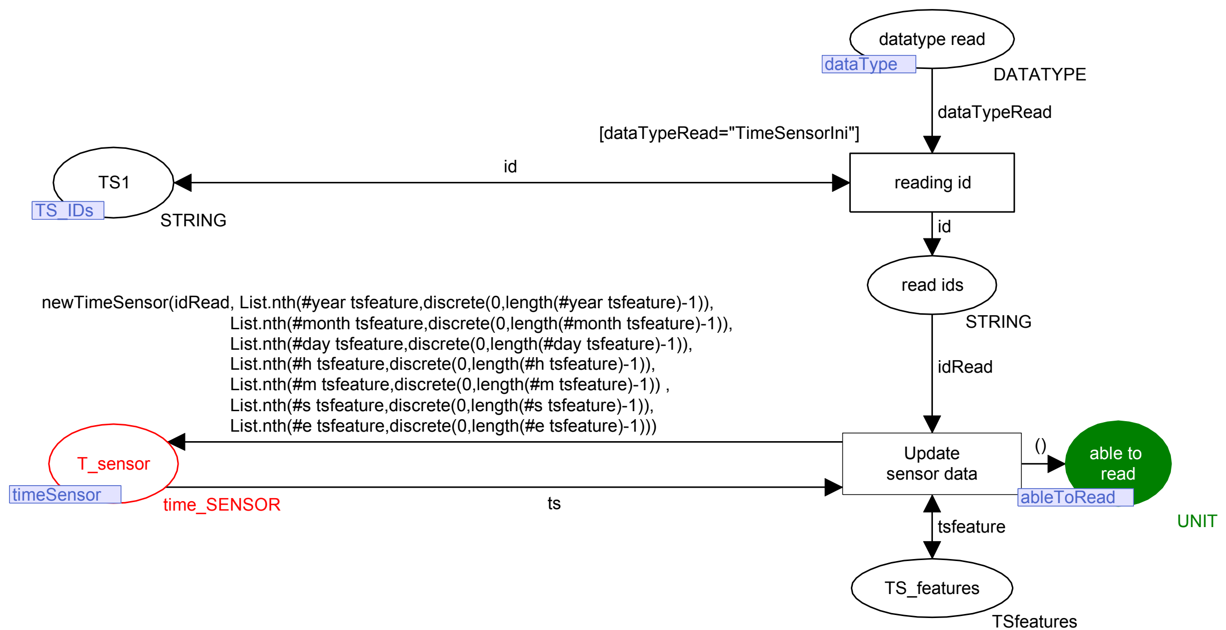

Figure B.0.3: *CPN analysis* time sensors update module

Appendix C: Evaluation Questionnaire

This questionnaire was used to evaluate the usefulness, satisfaction, ease to use and ease to learn of the APEX framework by software engineers who used it. Comments were also registered.

APEX questionnaire

		Strong disagree			...			Strong agree
Subject caracterization								
1	Age							
2	Gender							
3	Academic background							
		-3	-2	-1	0	1	2	3
4	Previous knowledge of Petri nets modelling							
5	Previous experience with Second Life/Opensimulator							
6	Software modelling experience							
7	Prototyping experience							
8	Familiarity with Ubiquitous Computing							
Usefulness								
9	It would help me visualize a design							
10	It would help me develop design alternatives more quickly							
11	It would help me evaluate design alternatives							
12	It would give me more control over the activities in design and develpoment							
13	It would make the comparision of different design alternatives easier to get done							
14	It would save me time							
15	It meets my needs							
16	It does everything I would expect it to do							
Ease of Use								
17	It is easy to use							
18	It is simple to use							
19	It is user friendly							
20	It requires the fewest steps possible to accomplish what I want to do with it							
21	It is flexible							
22	Using it is effortless							
23	I can use it without written instructions							
24	I don't notice any inconsistencies as I use it							
25	Both occasional and regular users would like it							
26	I can recover from mistakes quickly and easily							
27	I can use it successfully every time							
Ease of Learning								
28	I learned to use it quickly							
29	I easily remember how to use it							
30	It is easy to learn to use it							

Figure C.0.1: APEX questionnaire - first part

		-3	-2	-1	0	1	2	3
31	I quickly became skillful with it							
		Strong disagree			...		Strong agree	
		-3	-2	-1	0	1	2	3
Satisfaction								
32	I am satisfied with it							
33	I would recommend it to a friend							
34	It is fun to use							
35	It works the way I want it to work							
36	It is wonderful							
37	I feel I need to have it							
38	It is pleasant to use							
Open questions								
39	Indicate what you consider the strong and weak points of APEX							
40	Other comments							

Figure C.0.2: APEX questionnaire - second part

Appendix D: Exercises Proposed During the Evaluation

The goal of this exercise is to compare different solutions to the problem of creating a ubiquitous environment that guides elderly persons to the bathroom at night. Please create the models described and answer the questions.

1 - Basic setup (save the solution with the name *solution1.cpn*).

Current time: _________

Suggestions:

- Put lights on the floor to be turned on when the person leaves the bed;
- Put a presence sensor near the bed to detect when the user leaves the bed;
- Use functions *sendTurnLightOn(obj:OBJ)* and *sendTurnLightOffobj:OBJ)* in the CPN transitions in order to switch the light *on* and *off;*
- Use the function *nobodyNearPresenceSensor(ps:presence_SENSOR)* in the transition guard in order to enable the light turn off.

Current time: _________

1.1 - Is the developed solution adequate? What kind of problem can appear with this solution?

2 - Update the environment in order to support turning the light off only when the user reaches the bedroom. Save the solution with the name *solution2.cpn.*

Current time: __________

Suggestions:

- Use a second presence sensor

Current time: __________

2.1 - Is the developed solution adequate? What kind of problem can appear with this solution?

3 – Update the environment in order to use a timer to determine when the light can be turned off. Save the solution with the name *solution3.cpn.*

Current time: __________

Suggestions:

- Make use of the time sensor values (clone the *time_SENSOR* place into your model);
- Use the function *timeElapsed(ts:time_SENSOR)* to access the time elapsed of the time sensor *ts;*

Current time: __________

3.1 – What solution do you consider more adequate? Why?

Appendix E: APEX Brief User Guide

APEX can be used in developer and user modes. This appendix describes how to use APEX in both modes of usage and how to install it. The description uses the smart library example as illustration.

Installation

The Opensimulator server must be installed and appropriately configured (see build instructions at the Opensimulator website[52]). The CPN model is then loaded by the CPN Tools and executed. For execution to occur, a connection must be made with Opensimulator. Once the Opensimulator server is executed, the *communications/execution* DLL loaded, and connection to CPN Tools enabled, users can connect through viewers to the virtual environment by means of appropriate accounts that have been previously created in the Opensimulator server (*create user* command). Each avatar have an associated Opensimulator account. These steps are illustrated in Figure E.0.1.

User Mode

To use APEX, the user must:

1. *download* a viewer to connect to the virtual environment via the web;
2. *run* the viewer with the option *-loginuri http://A.B.C.D:9000/*, where *A.B.C.D* is the IP address of the server machine (external host name);
3. *connect* to the server machine using a provided user account.

It is assumed that the Opensimulator service is running in the server machine on port 9000.

Having taken these steps free exploration and interaction with the virtual environment is possible. In this mode the CPN *base model* should be complete and does not need to be extended. The virtual environment should be complete too. Command execution is achieved

[52] Opensimulator build instructions: http://opensimulator.org/wiki/Build_Instructions (last accessed: 22 February 2012)

with the viewer tool using features provided by APEX (e.g. initialize the synchronization between the simulation and the modelling layers). The commands provided are:

- *load oar filename* – load an environment or object from an oar file;
- *save oar filename* - save an environment or object from an oar file;
- *clear* – remove all objects from the scene. Optionally, to remove only one object a "!" must be added to the beginning of its name before using the *delete* option provided by the viewer;
- *obj ini* – send the *dynamic objects* values (*ID*, *type* and *position*) to the CPN model;
- *sensors ini* – makes the sensors initialization at modelling layer with the values of the sensors present in the environment;
- *sensors update* – performs sensors update with the current values;
- *addList:X* – add the value X to the user list (e.g. X is the book which the user wants);
- *remList:Y* – remove the value Y of the user list.

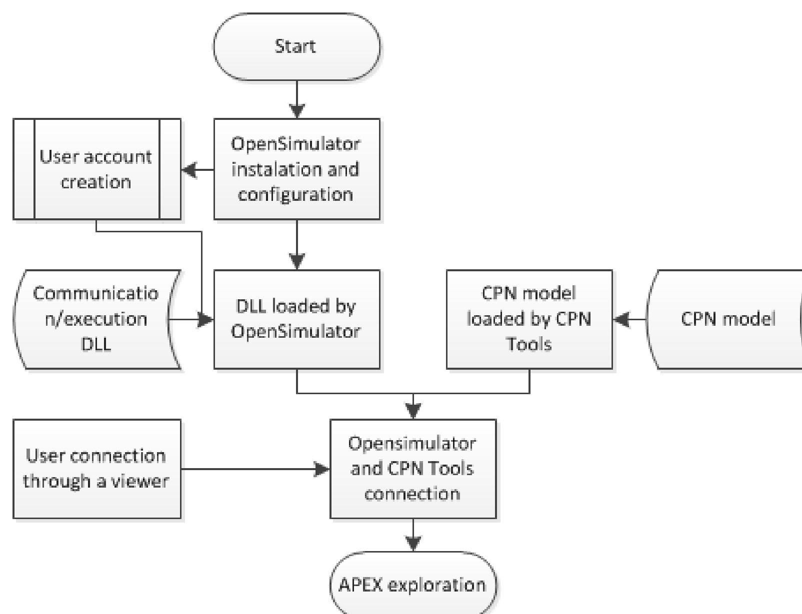

Figure E.0.1: Installation steps diagram

Developer Mode

In this mode the developer extends the CPN *base model* with the behavioural modules that specify the devices present in the virtual environment using the CPN Tools. The *base model* has information about the context of the environment that is used by the added modules. A

library of modules are available and used by the developers to model the ubicomp environment. The development of a new module is only necessary if it is not already available. Their development follow the process presented in Chapter 4. The virtual environment, if it does not yet exist, must be developed using a compatible viewer. Third party developed objects (*meshes*) can be imported using an appropriate viewer (e.g. Firestorm viewer) or created through it, see Section 5.2 for this purpose. To be possible to animate correctly an environment some environment conditions must be satisfied:

- Each *dynamic object* in the virtual environment (represented by a token in the CPN model) must:
    - have a unique ID present in the field *Name.* This ID is used to identify the objects in both layers (thus linking/associating a token to the correct object in the virtual environment);
    - indicate its object type using the field *Description (e.g. object type = gate)*;
- Each non-dynamic object must have the default name *Primitive* in the field *Name*;
- Each sensor must be loaded from the pre-defined sensors provided (OAR files). Alternatively new ones can be defined. The Figure E.0.2 illustrates sensor features. They are:
    - The fields *Name* and *Description* must be changed to reflect the desired values;
    - The *objectIDs* list present in the *Description* field of the *Presence Sensors* represents the *Ids* of the objects that the sensor affects. The elements of this list must be separated by commas ‘,’;
    - The *threshold* present in the *Description* field of the *Presence Sensors* represents the distance from which the sensor reacts;
    - The *value* present in the *Description* field of the *Light Sensors* represents the light value of which the sensor is exposed from 0 to 24 (day light correspondence). Modifications to this value will be reflected in the environment and modifications of the light in the environment will be updated to this value (the corresponding token in the CPN model will be continually updated).

Figure E.0.2: Sensor's attributes

The behaviour of the *dynamic objects* in the virtual environment is triggered by the associated module (connected by common token and object IDs) and made concrete by associated scripts. Figure E.0.3 shows a script associated with a *dynamic object*. In this example the script that determines the behaviour of the gate is linked to it by using the object script association that is provided by the viewer. When a gate is in the *open* CPN state, this state must be reflected in the environment. The script of Figure 3.5 (page 44) describes the behaviour of the gate.

This script is responsible for opening a sliding door when touched and to close it when touched again. The first touch changes to the *opened* state of the gate module and the second touch to the *closed* state (see Figure 4.2). The *touch action* present in the script (*touch_end*) is automatically triggered in the environment by the APEX *communication/execution* component and Opensimulator API when the relevant CPN transition is executed. To effect a model state change, one or more actions are executed in the environment responsible to it. This component searches for the objects in the environment that must change and triggers the *touch action* on the ones that it finds.

The developed script starts in the *default* state. It invokes the *touch_end* function that calls the auxiliary function *do_process* when the object associated with this script is *touched.* However, the touched actions are automatically triggered by the *communication/execution* component when the CPN model indicates that the gate should react (modification of state) without being touched by an avatar. The *counter* variable indicates the state of the gate (0 -

closed; 1 - open). The *do_process* function uses it to decide to open or close the gate. The *counter* is initialized with zero (gates should initially be closed). To open or to close the gate in the virtual environment the coordinate *X* of the gate is incremented/decremented with two values respectively (*llSetPos(llGetPos() +/- <2,0,0>)*. These cause the sliding of the door. The Figure 3.6 (sub-Section 3.1.3) illustrates the process that leads to the opening of the gate when a user arrives near it.

To associate a script to a device present in the environment the following steps must be carried out in the Second Life viewer:

- Request Admin Status: *Ctrl+Alt+G* (optional depending of the Opensimulator server version);
- Force the owner of the object to be the avatar currently connected: *Admin/Object/Force Owner To Me* (optional depending of the Opensimulator server version);
- Add the script: *Select the object/Edit/Content/New Script/Add the script/Save.*

After these steps APEX is ready to be used. At this stage the system assumes that the developer has accomplished the essential steps needed to use APEX (presented in previous section, *user mode* usage description).

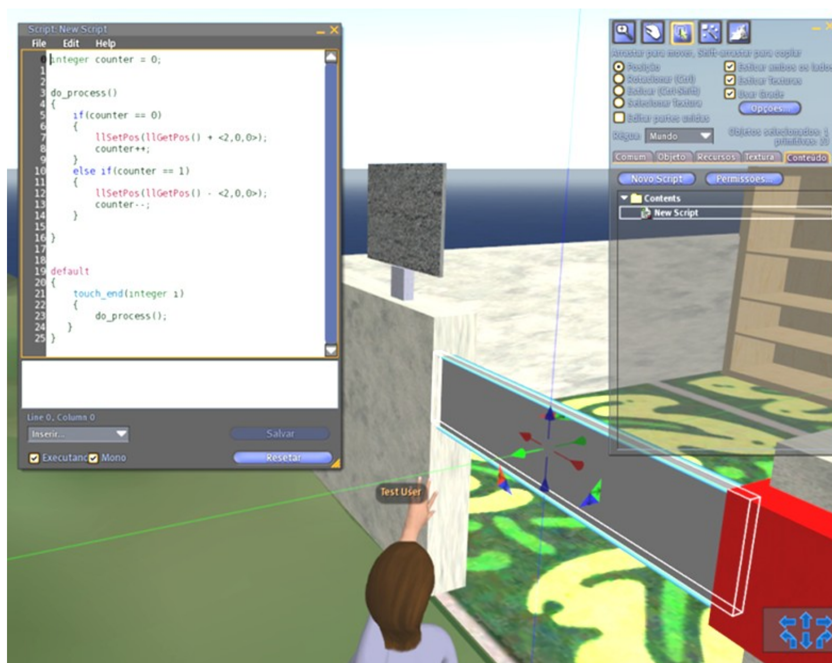

Figure E.0.3: Dynamic Object Script association

The *Time* and *Movement* sensors are invisible in the environment but their values are updated in the CPN model. The insertion of explicit sensors (e.g. presence sensor) in the world is possible using the command *load-oar filename* that permits sensor loading (see Figure E.0.4). Many sensors and objects can be used "off the shelf" as part of the environment when appropriate using this command. Also pre-defined environments can be loaded using this command. After the loading of the sensors in the world their features must be completed (e.g. *type*, *objectIDs* which it affects, *threshold*, etc.) using the field *description* provided by the viewer for each world object (see Figure E.0.2). These features are forwarded to the relevant token in the CPN model. This is achieved using the stated *sensors ini* command, invoked in the viewer. The sensor's features (e.g. *value*, *threshold*, *position*) can be modified in the viewer at runtime. This feature enables the development of different configurations of a ubiquitous environment at runtime. When the environment is ready, the *sensors ini* and/or *obj ini* commands must be executed in the chat box (see Figure E.0.4) in order to synchronize the CPN model with the new environment.

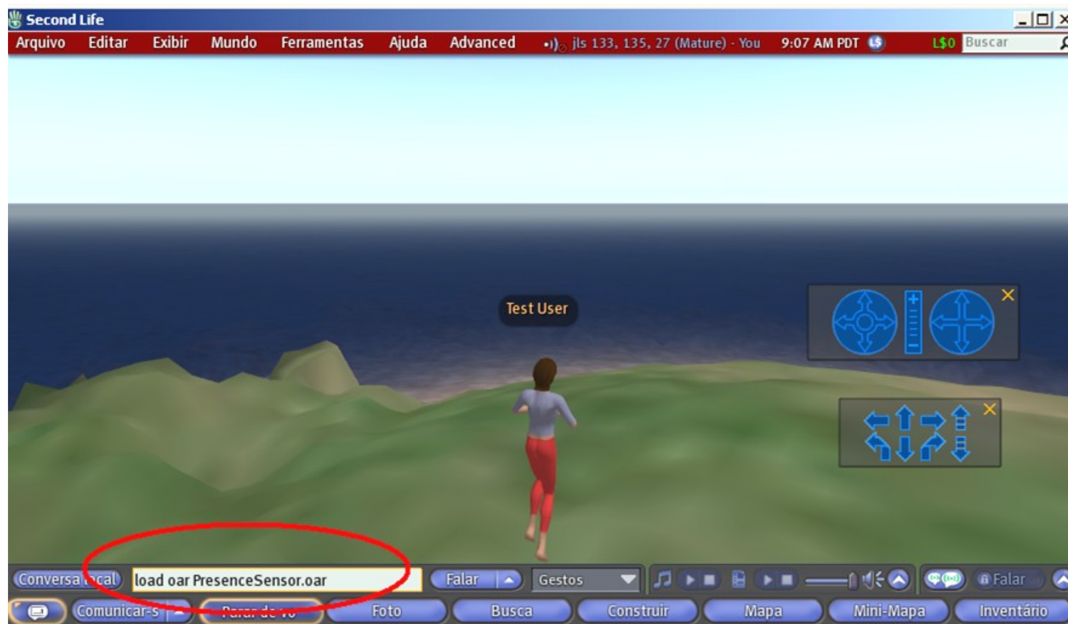

Figure E.0.4: Execution of commands in the viewer