Universidad Autónoma de Madrid
Escuela Politécnica Superior
Departamento de Ingeniería Informática

Doctor of Philosophy Dissertation

Easing the Smart Home: a rule–based language and multi–agent structure for end user development in Intelligent Environments

by

Manuel García-Herranz del Olmo

Co–advisor: Dr. Xavier Alamán Roldán
Co–advisor: Dr. Pablo Haya Coll

Madrid, Spain, European Union, 2009

A mis padres, Mary Carmen y Jaime, conspiradores, instigadores, compinches y mecenas. Y a Mila, compañera consciente y voluntaria.

Abstract

As computing, networking and sensoring technologies evolve, increasing their speed and lowering their size and cost, the number of computerized elements present in our daily lives is ingrowing steadily and fast both in number and diversity.

How many objects with processors, sensors or actuators do we already cross by in an ordinary day? How much information is retrieved somehow during the day about our lives? How many combinations of that information and capabilities may result which benefit us? To whom of us exactly?

Despite their evolving number and diversity, most computing enriched elements found in the environment are isolated from one another, created by different third party developers, strange to the end–users lives and particular needs. This is, in turn, a problem of control. Problem of special relevance in personal environments which are free–choice environments and strongly involved in families and individual self–definition.

Starting from this position, the present thesis analyses the problem of putting the end–users in the loop of deciding the behaviors of their own environments, analyzing, due to the heterogeneity of users, preferences and environments, the adequateness of expression and structuring mechanisms. What sort of language is best to deal with unskilled users with simple preferences while allowing to express highly complex behaviors? How can we deal with multiple inhabitants and domains of automation? What may happen when different users with conflicting preferences share the same environment? How can we benefit from automatic learning while preserving predictability? What sort of intelligence is fit to co–live with people in their personal spaces?

In response to these questions, an Event Condition Action (ECA) rule language is presented as the underlying kernel language for application–independent control. This language is designed as an UI–independent expression and explanation mechanism in Intelligent Environments, opening the door to predictable automatic learning. The naturalness of the language and its adequateness to deal with non–programmers has been studied through an end–user survey, measuring the resemblance of natural programming structures with those of the kernel language, while its complex expression capabilities have been compared with state of the art event composition algebras. In addition, a multi–agent structure is presented to organize and manage multi–preference environments, allowing users to translate their natural hierarchies in their enriched environments, replicating the different degrees of complexity present in human organizations.

The system presented in this work has been deployed and tested in various environments of different nature as well as in combination with other state of the art technologies. The particular challenges present in every environment, as well as the synergistic and wrapping capabilities of such a control mechanism extracted from our experience, are finally exposed.

A medida que evolucionan las tecnologías de computación, comunicación y sensado, aumentando su velocidad y reduciendo su tamaño y coste, el número de elementos computacionales presentes en nuestra vida cotidiana se está incrementando rápidamente, tanto en número como en diversidad.

¿Cuántos objetos con procesadores, sensores o actuadores nos cruzamos en un día cualquiera? ¿Cuánta información acerca de nuestras vidas es obtenida por estos elementos a lo largo del día? ¿Cuántas combinaciones de esa información y las capacidades de esos elementos podrían redundar en nuestro beneficio? ¿En beneficio de quién de nosotros, exactamente?

A pesar de que los elementos con capacidades computacionales son cada vez más numerosos y variados en nuestro entorno, la mayor parte de ellos conviven aislados unos de otros, creados por desarrolladores profesionales, extraños a nuestras vidas y necesidades particulares, lo que redunda en una incapacidad de control por parte del usuario. Este problema es de especial relevancia en entornos personales, en los que la elección de las metas y métodos depende de sus usuarios y en los que el entorno participa en la definición que de sí mismos hacen sus habitantes de manera especialmente cercana.

Partiendo de estas premisas, la presente tesis analiza el problema de introducir al usuario final en el proceso de decisión de comportamientos de su entorno, analizando la adecuación de distintos mecanismos de expresión y estructuración a la diversidad y heterogeneidad de posibles usuarios. ¿Qué clase de lenguaje es capaz de permitir expresar preferencias simples a usuarios con bajos o nulos conocimientos de programación al mismo tiempo que permite expresar preferencias complejas a usuarios avanzados? ¿Cómo podemos afrontar múltiples usuarios y dominios de automatización? ¿Qué debe ocurrir en el caso de que usuarios con preferencias opuestas compartan entorno? ¿Cómo podemos sacar partido del aprendizaje automático sin afectar a las expectativas y previsiones que un usuario tiene de su entorno? ¿Qué clase de inteligencia es adecuada para convivir con nosotros en nuestros entornos privados?

En respuesta a estas cuestiones, esta tesis propone un lenguaje basado en reglas ECA (Eventos Condiciones Acciones) como lenguaje base para un control integral, independiente de la aplicación que se quiera controlar. Este lenguaje ha sido diseñado como un mecanismo para la expresión y explicación de comportamientos de Entornos Inteligentes, independiente de la interfaz de programación utilizada, abriendo la puerta a un aprendizaje automático predecible. La naturalidad del lenguaje, así como su adecuación a las capacidades de expresión de usuarios sin conocimientos de programación, ha sido analizada mediante un estudio de usuario, probando las semejanzas del lenguaje con los mecanismos naturales de programación humanos. Por otra parte, su capacidad para expresar comportamientos complejos ha sido comparada con álgebras de composición de eventos del estado del arte. Adicionalmente, esta tesis propone una estructura multiagente que permite organizar y gestionar múltiples preferencias en un mismo entorno, permitiendo a los usuarios del mismo trasladar sus jerarquías naturales a los Entornos Inteligentes, replicando los distintos grados de complejidad presentes en las

estructuras sociales.

El sistema descrito en esta tesis ha sido desplegado y probado en varios entornos de distinta naturaleza así como en combinación con otras tecnologías del estado del arte por lo que, finalmente, esta tesis resume los particulares retos que presenta cada entorno, así como las sinergias observadas en nuestra experiencia.

Acknowledgements

Existe en mi trabajo la filosofía subyacente de que la base, objetivo y medio de todo sistema han de ser las personas. La tecnología, por tanto, ha de ser convertida en una herramienta humanizadora. Un canal capaz de extender las capacidades de las personas más allá de sus límites actuales sin desvirtuar su esencia humana más profunda. Lejos de ser casual, esta filosofía y este trabajo surge de la afortunada experiencia vital y humana que he tenido y tengo la suerte de disfrutar, fruto de la dichosa coincidencia de grandes personas –de su esfuerzo y de su fe– a las que quiero y debo agradecer.

En primer lugar a mi familia, y en especial a mis padres, tesoro que me ha enseñado el valor del amor, las ideas, la excelencia, la libertad y la convivencia sin ocultar el precio que por todo ello se debe pagar. Por regalarme una cultura y un amor por la cultura impagable y por descubrirme "el misterio de Dios en la única realidad existente". Os lo debo todo.

A Xavier Alamán, siempre abriendo puertas, sin cuya confianza, enseñanzas y apoyo no habría tenido ni por dónde empezar. A Pablo Haya, guía y compañero al que le debo tanto de este trabajo y de mi experiencia doctoral, por hacerlo siempre personal y por ser siempre más que los demás en su búsqueda de ser más con los demás. A Germán Montoro, por su apoyo y desinteresados consejos y al resto de componentes de AmILab, crisol de este trabajo, cuyo funcionamiento y ambiente se debe a la aportación individual de humor y rigor de cada uno de sus miembros. A Manuel Freire por su generosa ayuda, a Manuel Cebrián, cuya amistad y agradables charlas me han acompañado y ayudado durante esta tesis y a todos los miembros de mi departamento, en especial al grupo GHIA.

A Alexandra Cristea, Gladys Castillo Jordán, Juan Carlos Augusto y Alvaro Ortigosa por sus revisiones, comentarios y sugerencias y a todos aquellos colegas que con su esfuerzo, consejos y conversaciones me han ayudado consciente o inconscientemente a lo largo de esta tesis, entre los que se encuentran Alvaro García, Ana Iriarte, Roberto Moriyon, Ivan Dotu, Dominik Schmidt, Fabian Hugelshofer, David Molyneaux, Rose Gostner, Yukang Guo y Eran Eden. Y, especialmente, a Juana Calle, Marisa Moreno y Sonia Durán sin cuya delicada

dedicación hubiera acabado loco hace tiempo.

A mis amigos, con los que tengo la suerte de crecer y enriquecerme y que me han apoyado tanto moral como físicamente en la realización de esta tesis. Gracias de todo corazón.

A la responsable de la biblioteca de la Fundación Ortega y Gasset, cuyos periódicos envíos de material humanista y científico, han servido de base a más de medio capítulo de esta tesis.

Y, finalmente, a Mila, por bregar conmigo y apoyarme incondicionalmente. Gracias Mila.

Gracias a todos.

Contents

Chapter 1

Introduction

> *"Therefore we are trying to conceive a new way of thinking about computers in the world, one that takes into account the natural human environment and allows the computers themselves to vanish into the background"* [122]

Almost twenty years after Weiser wrote of a new way of thinking about computers, Ubiquitous Computing has become an adolescent discipline with many fields of study, wide-ranging interests and vast possibilities. In fact, few disciplines are as holistic and multidisciplinary as the one triggered by Weiser's vision and this is, in my opinion, its biggest beauty and its greatest challenge.

Ubiquitous Computing's pubescence uncovered the main challenge of our discipline: "vanishing computers". Soon it was realized that this was a twofold challenge: **Physical** — in which computers as big boxes in a table are to be removed and, instead, embedded into everyday-life objects— and **Conceptual** — in which what is to be removed is the specific knowledge needed to use a computer, just as nobody needs to know today anything about electricity generation to plug a toaster in the kitchen. This physical vanishing required smaller processors, lower energy consumption devices, greater network bandwidth [123], more varied sensors and other hardware improvements to embed sensing and communicating computers in everyday-life objects. Conceptual vanishing, on the other hand, required new interaction mechanisms for computers distributed in many objects (traditionally not connected with computers) but in the paradoxical absence of a physical computer. Graphical, oral, physical, emotional and other interfaces from the Human Computer Interaction community were devised for or applied to the field.

Some of Ubiquitous Computing's first steps have already found their way to the market and into our lives. Traditional computing entities such as mo-

bile phones, portable music players or game stations are becoming smaller, more powerful and cheaper (Moore was right). In addition, they sense and react to many *more stimuli*. First, Graphical User Interfaces (GUI) widgets, as the traditional way of interaction, are being enhanced and extended through the use of multi-touch devices. Second, many more interaction paradigms are being integrated in devices: phones make calls based on natural speech interfaces, mp3s shuffle their songs when they are shaken, mobile phones are silenced when turned face down on a table and, in some game stations, batting a ball requires no buttons or pads (just a good swing). These new interaction possibilities respond in some degree to the also *increasing amount of services* those small computers provide: agenda, calendar, notes, contacts, tasks, multimedia players, phone calling, pagers, GPS navigators, cameras or even projectors come together or in sets in those small devices (which are becoming increasingly harder to define as PDAs or mobile phones). In any case, the growing omnipresence of Internet access is gradually transforming them into virtually anything/everything. As a result, the potential of those devices is spreading so far that they have to come in different packets, a pret-a-porter solution for the technology market multiplying the user's possibilities even more: small mp3s with no screens for joggers, big ones with a high capacity for home-lovers, medium for travelers; mobile phones for teenagers, simplified versions for the elderly, expanded versions for businessmen and so on. Each one with its own capabilities and interaction standards in what is becoming everything but a global experience "as refreshing as taking a walk in the woods" [122].

But this is just the beginning, and the greatest implications of this computing vision are yet to come. Firstly, the amount of computers present in the environment will grow exponentially. Besides obvious computers such as PCs, PDAs or mobile phones, the environment will have many more "computers" in fridges, mugs, windows, lamps, switches, walls, shoes, pens, pans, cupboards, taps, stoves, doors, mats, notes and many other objects distributed throughout our traditional environment. Therefore, the first consequence of this vision is the **number**: computers are not only vanishing into the background but multiplying exponentially backstage. Secondly, since computers will be embedded in everyday-life objects, they will be present in every space/scenario and will have to support different users in different tasks through different tools. Most importantly, since they will be present everywhere, users will be forced to deal with them —as we have to deal today with writing or electricity. Thus, the second implication of the Ubiquitous Computing vision is **diversity**: spreading and multiplying computers in the background also multiplies the number of users and tasks required to be "computerized".

Returning to Ubiquitous Computing in today's market, besides those traditional computing devices, we can already find some of these new computational

entities in our lives. The examples, even in this nascent stage, are already wide and varied: RFID tags are found in supermarkets and shops, in car keys, access cards or warehouses; Nike^{TM}is selling shoes with sensors that communicate with Apples's iPod^{TM}, lighting automation with movement sensors is required by law in some public places and many other sensors and actuators are distributed along our natural environment for different purposes (e.g. security alarms, video surveillance, irrigation, heating or health care), not to mention the growing supply of automation and control systems for domains as varied as high-standard homes, factories, hospitals or government institutions. As Weiser believed in 1991, the performance of technology is reaching the level to meet with his Ubiquitous Computing vision. Can we say the same for applications?

We can realize the impact of *diversity* in the applications domain when we look at today's computers. Many of them run the same operating system and almost every one of them is based on the same screen-windows-mouse interaction paradigm. Even though this is beginning to change, most applications are still designed with a specific purpose. Most importantly, users tend to use applications just for the purpose intended by the designer —with small exceptions such as using mail applications to transfer files or the tray clock configuration menu as a calendar. In other words, despite its relative expansion, computing is still quite homogeneous: i.e. restricted to the computer domain, most computing user–oriented processes share paradigms, tools and uses. Ubiquitous Computing, on the other hand, aims precisely to break the "computing-computer" association and transform it into "computing-everywhere", breaking, among other things, the classical homogeneity of computing processes. Real spaces do not have to share strong interaction paradigms — such as the the mouse and the windows in computers— and their objects may be varied and used in many different ways —from fridges or TVs as lighting devices, ovens as pan cupboards or beds as reading places, to give an example from the home domain. Consequently, Ubiquitous Computing applications will need to deal not only with the high number of computers present in the environment, but also with the combination of users, goals and tools.

An example of this diversity and a glimpse of its consequences can be seen nowadays in market applications that, even though far from ubiquitous, are already outside the computer in some way. I will expose a personal example: I have a watch that wakes me up in a range of 20–30 minutes according to the sleep phase, choosing the least traumatic moment. My toaster is able to detect the color of the toast so the bread is perfectly toasted. My bedside table alarm clock knows the temperature and humidity of both the inside and outside of the house and can make basic weather predictions based on them. The heater and irrigation in my home can both be programmed and the oven can clean itself via a pyrolytic process. My TV is able to record any program and store it in

"my channel" for me to watch at anytime. Additionally, my mobile phone has a GPS. In some sense I can say that my environment is able to perceive: the mobile phone knows where I am, the alarm clock what the weather is like, my watch how deeply asleep I am or the toaster how toasted the bread is. Additionally, they are able to self-actuate: the oven can clean itself, the TV can record programs and the irrigation and heater can be turned on and off automatically. Those are the cornerstones of Ubiquitous Computing, but I strongly feel that the essence is still missing. This is not because of the individual capabilities of the environmental elements, but because of how the environment works as a whole. Waking me up earlier if the weather prediction shows strong rains (and consequently the traffic will be worse) or synchronizing the alarm clock with my watch (so I'm not only woken up at the right time but also with nice music instead of an annoying buzz) are impossible things, even though the necessary information and capabilities are provided by one device or another. This is the first hole: **isolation, meaning unshared information/capabilities** (e.g. How many clocks do I have to update every time there is a blackout? Can't they really synchronize themselves?). The second hole has to do not with the capabilities of the environment but with the way I am supposed to interact with it. Programming my alarm clock requires turning a wheel, pressing two buttons (one increases the time in 5–minute intervals, the other decreases it in 1–minute intervals) and shifting some levers to choose between radio, buzzer or no alarm. Programming the alarm on my watch, on the other hand (the left one, actually), requires keeping a button pressed for 2 seconds, then pressing a second button to enter a menu and set the time. There are no levers and I actually do not know how to deactivate the alarm so it has been sounding every day since I bought it. The irrigation is conceptually similar to the alarm with an additional "duration" concept (turn on at 9:00 am for 2 hours) but programming it is completely different; there is a dial surrounded by small buttons in circle, simulating a 24–hour clock in which every small button corresponds to 30 minutes (the top button corresponds to the 00:00 to 00:30 time interval). If the button is pressed the irrigation will be turned on those 30 minutes. Conversely, the irrigation in my parents' house has a new display with completely different buttons, menus and options. Different from my dial programming system, to that for my watch, to that for the alarm clock and (I am sure this is no surprise) to that for my TV. Thus, for the concept "at *some time* do *something* for *this* long", I had to learn 5 different interfaces for 5 different devices. Sadly, I am fairly sure that if I acquire a sixth device I will be acquiring a sixth interface too since, as a user, I am inexorably bound to each of the designers' interaction preferences. In conclusion, a second hole is revealed: **unsoundness or unshared interaction mechanisms and places** (i.e. every device has its own interaction mechanisms and has to be used from a different place).

In these 20 years, the research community has been exploiting the many possibilities of environments and objects capable of perceiving, computing, communicating and actuating. This research has lead to many context–aware applications often referred to as *Smart*: Smart kitchens, Smart floors, Smart laundries, Smart mugs, Smart offices, Smart mirrors and more. This diversity of Smart things is a potential victim of the two holes posed previously, with the obvious inconvenience that they will be everywhere. Especially because of their numbers, *Which device was able to perceive what?* and *How do I use that device?* are questions to be dismissed if we want Ubiquitous Computing to be "as refreshing as taking a walk in the woods". Can you imagine posing those questions with electricity? Which socket was able to charge what device? How should we plug our devices in a specific socket to be charged? I am, personally, sure Weiser would not have considered it a disappearing technology in such a case.

1.1 Background

As noted previously, having different interaction mechanisms and places for every device is beginning to be a problem even today, with few computing devices with which to interact compared to the futuristic Ubiquitous Computing scenario with thousands of computing entities present in the environment. This is, in summary, a problem of **control**. How can users control their environment?

This question has been the main focus of research over the last 10 years at the Ambient Intelligence Laboratory (from now on referred to as AmILab) at the Universidad Autónoma of Madrid. The first problem was that of **abstraction**. Strongly entangled to the *isolation hole* revealed above, it involves the challenge of perceiving the numerous and varied computing entities in the environment in a homogeneous way. Interaction was not addressed at the time, but a mechanism — a blackboard-based middleware — was settled upon for building context-aware global applications. That is, my irrigation system will never be alone again and applications can be built to combine its capabilities with the weather information retrieved by the bedside table clock. This middleware (from now on referred to as **the Blackboard**) is a simple representation of the environment in terms of *entities* of a *type* with *properties* and *relations* with each other (e.g. "grandma's lamp" of type "light" with property "status" and the relations "located at" linking it with the entity "kitchen" of type "room" and "belongs to" linking it with the entity "mother" of type "person"). This representation is guided by a public schema so abstraction is easily accessible to any application. Access, on the other hand, is done by a simple API with three main functions: *Get*, *Set* and *Subscribe* for retrieving, modifying and being notified of changes, respectively, in the Blackboard elements. Thus, applications can work with any element of the Blackboard in the same manner, regardless of the low–level details of the real element: e.g.

a KNX [1] light or a Phidget [52] controlled light will be both turned on if the status of their Blackboard representation is set to on. This is done thanks to a set of drivers translating each technology to the Blackboard representation, but that is beyond the scope of this work and, even though we will have to go back to it at some point, it is better described in [58][59].

Once the problem of perception (i.e. the *isolation* hole pointed above) is addressed, the problem of human interaction (i.e. the *unsoundness* hole or unshared interaction mechanisms and places problem) was the next natural step in our path. The Blackboard provided the necessary tools not only for creating one–for–all interfaces, but also for generating them automatically and expanding their capabilities through context awareness. Two interfaces, developed by AmI-Lab group members, give an example of these possibilities. First, a GUI, formerly known as Jeoffrey [2][61] and then improved as iFaces [50] uses the Blackboard information to automatically generate a graphical user interface for every environment. That is, the elements present in the room, their position, associated images and capabilities are extracted from the Blackboard and the scheme to, for example, present "grandma's light" in the right corner of the kitchen and with a slider in its control panel (since it is an adjustable light). Using this information, iFaces can also adapt the interface automatically to different kinds of displays, e.g. knowing that an "adjustable" element will need some kind of slider and that touch screens need large buttons, well suited to being touched by fingers, it generates a different interface for them than for PDAs. The second interface example is Odisea [85][86][87], a natural language interface with a corpus and grammar that are also automatically generated from the Blackboard's information. Thus, any element added to the environment will automatically be present in both iFaces and Odisea (the graphical and natural language interfaces, respectively). Additionally, using the information from the Blackboard, Odisea can disambiguate commands by checking the context state. For example, if the user says "turn off the light" (or the recognizer only recognizes that), Odisea can check how many lights are turned on in the environment, asking the user which one of them to turn off or turning it off if only one is on.

1.2 Challenges

> *The future world of ubiquitous computing is one in which we will be surrounded by an ever-richer set of networked devices and services. In such a world, we cannot expect to have available to us specific applications that allow us to accomplish every conceivable combination of devices that we might wish [96]*

Intelligent Environments (IE) are one of the consequences of Ambient Intelligence. Half way between Robotics, Artificial Intelligence and Human Computer

Interaction, they present a multi–disciplinary domain in which Ubiquitous Computing has to deal with already existing environments in which people are used to live in and have defined a status quo between the unintelligent physical places and their personal needs.

As analyzed by Cook, Augusto and Jakkula in an extensive survey [24], Ambient Intelligence is a discipline facing many challenges, from sensing, reasoning (e.g. modeling, activity prediction and recognition, decision making or spatial and temporal reasoning) or acting, to more human–centered challenges such as HCI or privacy and security issues in a wide range of applications (e.g. smart homes, health and monitoring assistance, hospitals, transportation, emergency services, education or workspaces).

Personal environments, in particular, cohere the most singular challenges of the status quo established between people and their environments. They play a role in group and individual self–definition [28], they are free–choice environments [72] and, in summary, they are built and used to deal with all the five levels of Maslow's hierarchy of needs [80], bottom up: physiological, security, social, esteem and self–actualization (see Figure 1.1).

Figure 1.1: Maslow's hierarchy of needs [80]. Represented as a pyramid, needs become more sophisticated as we move upwards in the pyramid. According to Maslow, new needs arise as we satisfy all the needs of the underlying levels.

Every need requires an specific solution but, as we travel higher in Maslow's pyramid, the solutions to satisfy these needs become more entangled with the person herself (e.g. while security can be handled by third parties, self–actualization requires a committed individual). Therefore, the environment's *Intelligence* has to be carefully balanced with it's *Idiosyncrasy*, which is a reflection of that of its inhabitants and the roles they play in them. The purpose of the intelligence, and the type of intelligence fit for that purpose are, consequently, two important aspects to consider.

From creating/manipulating tools to shaping the environment, humans have used and increased their control over the environment to fulfill their needs. In this sense, IE pose an interesting challenge: Being populated with multiple and interconnected sensors, actuators and computational capabilities, they provide the tools to enhance traditional environments with new ways of interaction and control: context–aware applications.

Context–aware applications are nothing more (or less) than indirect control mechanisms i.e. part of the control (or all of it) is commanded not directly by the user but indirectly through the context. Applied to the home domain, we found a direct way to build context–aware applications —through the Blackboard (see Section 1.1)— such as a photo frame showing the pictures belonging to the inhabitants in the room. Needless to say the inhabitants, the photo frame, the pictures and the room are represented in the Blackboard, as are their relations (i.e. that person "likes" that picture or is "located in" that room). These kinds of programs are designed by professional programmers to deal with particular problems or preferences with the degree of personalization they considered at the time of programming. In the photo frame example new pictures could be incorporated to the carousel by simply adding them to the Blackboard and relating them to the correct person. In response, the photo frame automatically incorporates the new picture to the picture carousel.

But while context–aware applications may enhance the control of and interaction with our environment the control problem is now moved to them: How can I control which applications run on my environment and what they do? Let us consider a user who prefers some pictures in the mornings, others in the afternoon, does not want some pictures to be shown when some other person is present and prefers to turn the carousel off when watching a movie. Similarly to what happens with technologies nowadays, even though the capabilities and tools are available in the environment, the user will find himself at a dead end: if the professional programmer did not think about it from the start, it would be impossible for the user to control the environment the way she wants. In other words, even though these applications will automatically adapt to some changes (such as the preferred pictures) their goals and means are fixed and may or may not be those the end–users need since they are out of the loop.

When we look at traditional computer applications, we find that users control which applications run on their computers by installing/uninstalling them, being forced to find a developer who thought of their problems before them. In addition, to allow users to control what the applications do, most developers provide their programs with some end–user configurable variables. This approach (though commonly used) has two main drawbacks directly related to the **unshared interaction mechanism** problem posed before: firstly, since different applications have different variables of interest, each of them will have

a different personalization interface, **making the personalization problem application–dependent**. Secondly, since the parameterizable variables must be chosen a priori by engineers, the more flexible they want to make the program the more variables they have to add to the interface, creating an **inverse relationship between flexibility and simplicity**.

These two drawbacks are at the core of a "disappearing technology": Naturalness and Easiness. Understanding naturalness as the way in which a system preserves the basic human structures and methods, and easiness as the simplicity of use it presents, i.e. preserving the *status quo* and having an accessible learning curve. In a way we are not proposing anything different from P. Maes [78] when she stated competence and trust as the two main challenges software agents need to solve. Thus, while an accessible learning curve empowers competence, preserving the status quo guarantees trust.

In addition, people has intuition more strongly developed in traditional environments than in computer environments, since they have lived much longer with the former than the later. They have not only a stronger preconception of how a mug is used than of how a web browser is used, but this preconception is so strong that they will tend to perceive a mug not working the way they presumed as a broken mug.

Therefore, as stated previously, "Which device perceive what?" and "How is that device used?" are questions to be dismissed in Ubiquitous Computing. Questions directly related to control that are hardly acceptable when considering them in any other "disappeared technology" such as electricity (e.g. Which socket was able to charge what device? How should devices be plugged in an specific socket to be charged?)

With these principles in mind, the most important challenge is **to put the end–user in the loop of indirect control applications**. That is, to transform programmed indirect applications into programmable indirect applications that allow end–users to program their own solutions.

Far from saying that professional applications cannot satisfy the end–user we argue that, while they are especially well–suited to dealing with common, transparent or complex problems such as finding objects, saving energy or managing security, they lack the flexibility to deal with most of the small personal preferences of users' daily lives. Extending Newman's et al. philosophy to indirect control, we believe that *users will wish to create configurations and combinations of these devices, and will likely want to create particular configurations that no application developer has foreseen* [96].

Hence, context–aware applications are the corner stone of IE. However, the main challenge of IE can be formulated as "how can the inhabitants of the environment control that applications" and how can we **provide environments with an intelligence whose purpose is to allow users to control their**

environments in order to fulfill their needs. This statement can be analyzed as four different challenges.

1.2.1 Living with another intelligence

When we talk about IE it is important to stress that the environment's intelligence must not only be chosen to deal with users' preferences but also to co–live with them in an environment that, as pointed out by Davidoff, plays a role in group and individual self–definition [28]. Imagine an aware house, able to actuate and to understand human preferences and desires. Considering Minsky's six–level model of mind [83], top to bottom: self–conscious reflection, self–reflection thinking, reflective thinking, deliberative thinking, learned reactions, and instinctive reactions (see Figure 1.2); What kind of intelligence would we desire the house to have? In which level of mind would we want it to stop? *2001: a space odyssey* (counting with Minsky as advisor) presented a fiction in which the environment's intelligence catastrophically reached the self–conscious level, one of the five dimensions of Salovey and Mayer's *emotional intelligence* [82] (i.e. self–conscious reflection, emotion control, motivation and self–motivation, empathy and social abilities). Emotional Intelligence subsumes Gardner's [48] concepts of interpersonal and intrapersonal intelligence, that is, the intelligence to deal with others and intelligence to deal with our own intelligence (and the problems it poses). While the former may be desired for many computing applications (see Cai's deliberation on Instinctive Computing for a deep view on the matter [17]), the latter can hardly be considered as acceptable in anybody's home, unless we are willing to have a possibly unmotivated, apathetic or selfish home.

Thus, going back to Minsky's model, IE should not provide environments with intelligence in every level but it should fade intelligence as it reaches the higher levels of the model. Therefore, considering the objective of "putting the end–users in the loop", the first step for IE is to **provide the means for users to program the "instinctive reactions" of the environment**. Based on them, some mechanism for learning reactions and deliberative thinking may be provided, carefully considering what may be learned and what may not. Reflective and self–reflection thinking must be particularly treated to deal with specific tasks such as self–maintenance and, finally, self–conscious reflection can be considered unfit to co–live with humans in personal environments.

1.2.2 Independent programming

Allowing users to program the instinctive reactions of an environment is strictly related to what they are to program. It is, in summary, a problem of interaction with the environment. The "unshared interaction mechanisms" problem, posed at the beginning of this thesis, is traditionally located in the application–

Figure 1.2: Minsky's six–level model of mind [83], as we move up through levels, intelligence becomes more complex, able to learn, to reason and so on until it is finally conscious of its own existence and intelligence and, consequently, work upon that.

dependent problem but, when faced with an environment rich in interaction capabilities, in which everything can be accessed from many places and, consequently, through different UIs, the application–dependent problem is extended to a UI–dependency problem: if the solution for a problem depends on the UI used to solve it, the increasing number of interfaces that become available (e.g. natural language, tangible interfaces or mobile phones) will bring the same drawbacks as the application–dependent problem posed above. Thus, **the programming method must be UI–free** and UIs must be just shortcuts to create the same control structures. In this way, UIs will be chosen according to the interaction needs of the moment and not according to their programming capabilities.

UI–independence requires thinking about **a kernel programming language** in which sentences or programs are created through a diversity of UIs designed to deal not with programming issues but to cope with interface needs. Thus, any programming structure can be created or modified by any UI (see Figure 1.3).

This kernel language should be **as close as possible to the end–user's way of thinking and programming** since, as stated by Myers, "the closer the language is to the end–user's original mental plan, the easier the refinement process would be" [93]. Additionally, creating a kernel language that takes into account the end–user's mental plan allows UI designers to focus just on interface issues and avoids having multiple programming paradigms — the UI–dependency problem —, one for each alternative a UI designer thought appropriate for the end–user to consider and program with that particular interface.

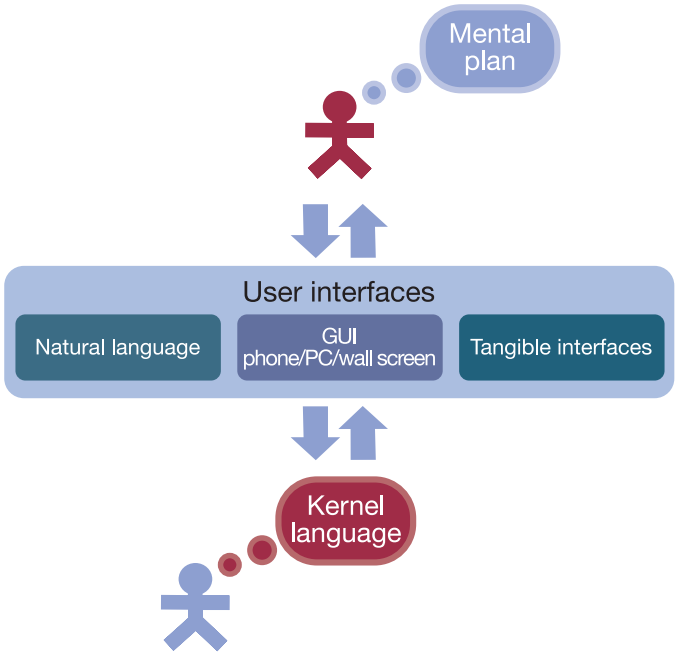


Figure 1.3: Programming flow, from the user to the kernel language and from this back to the user through a variety of UIs. UIs act exclusively as an intermediate interface layer between the end–user and the common logic kernel.

1.2.3 The end–user as a programmer

Considering end–users as the programmers brings some additional challenges, in particular when looking at their various degrees (or absolute lack) of expertise, their diversity of backgrounds and a completely different concept of commitment than that of a professional programmer. Firstly, the absence of a strict feeling of commitment makes end–users more inclined to abandon under frustrating conditions, thus, any alternative designed for end–users must **provide an increasing degree of complexity** with a very low starting point. Any improvement should stand on previous knowledge as in Papert's ideal of "low threshold no ceiling" [101] in which the difficulty of the system (or the challenges it poses) grows

according to the end–user's skills in order to avoid anxiety (when challenges surpass the skills) and boredom (when skills surpass the challenges) keeping the user in the optimal flow of motivation (see Figure 1.4). Secondly, the diversity of backgrounds, as stated at the beginning of this section, leads to many different potential goals among users and scenarios. Thus, the programming language must be **flexible** enough to cope with the different **expression** needs of each circumstance. This flexibility must be balanced with the **simplicity** required to deal with inexperienced users.

End–users, as inexperienced programmers, will likely tend to program their preferences as they come, without an overall design or a proper test bench, thus it may not be surprising that their creations are not as appropriate as they thought them to be. This problem can be addressed through two different tools: first, some sort of **explanation mechanism** will allows them to understand the insights of the environment's behaviors and its possible faults, thus a natural explanation to a "why did you..." question will provide a sort of debugging mechanism to end–users. Secondly, **automatic learning** may be applied to either adapt the end–users' programs to what they really wanted to program or to spot in advance the possible failure points of a program so as to ease the debugging process.

In summary, a system for end–user programming must balance simplicity (i.e. naturalness and easiness) and flexibility of expression, with progressive complexity, and provide scrutability and (semi)automatic adaptation of the end–users' programs.

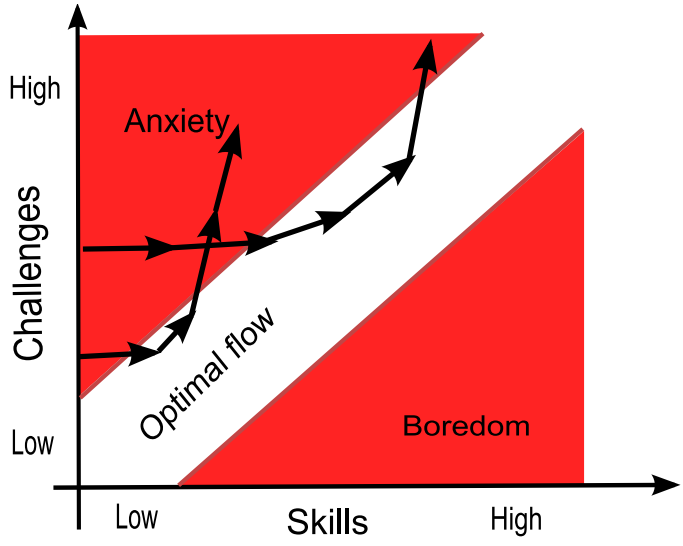


Figure 1.4: Csikszentmihalyi's [27] notion of flow to analyze motivational factors in learning. Used by Repenning and Iaonnidou as a guide for EUD usability [106] from which this picture is taken.

1.2.4 Idiosyncrasy of end–users' environments

Finally, end–user programming in Intelligent Environments present certain additional challenges in relation to the nature of personal environments. Firstly, each person may create different programs, with different goals and for different purposes. Even though these programs are created through the same mechanism, a failure in one of them must not be perceived by the user as a failure of the overall system, but just in the specific domain of the malfunctioning program, **maintaining the user's trust in the system when a part fails**. Additionally, personal environments may be populated with **multiple persons whose preferences and goals may conflict**. All these programs or preferences (from the same or different users) must, additionally, coexist in an environment in which a manager figure is not always clear (or not present at all). Finally, as stated by Davidoff, personal environments are more than just places and **"participate in the construction of family identity"**[29]. In conclusion, the programming structure must provide an easy means to distribute and manage programs as responsibilities, as well as the appropriate mechanism to deal with the various hierarchies and conflicts of a not necessarily centralized, multi–user scenario. Thus, the concept of human responsibility, as well as other hierarchical characteristics such as purpose or location, must be translated to the programming domain and tools must be provided for conflict resolution. Additionally, the creation and automation procedures must be directly controlled by the users in such a way that they can preserve those particular aspects of their lives they consider as fundamental to defining their identities.

1.3 Research contributions

This work presents a working solution to end–user programmable indirect control. The above challenges are addressed and handled through what can be seen as a twofold strategy: first, by **designing a language** that is able to deal with most of the requirements derived from having an end–user as a programmer (such as application–independent programming, an increasing degree of complexity, a simple and flexible mechanism of expression, means of explanation or automatic learning); and secondly, by **establishing an underlying structure** that, while complementing the language to deal with the rest of the end–user requirements (e.g. maintaining the user's trust in the system when a part of it fails), tackles those requirements derived from the idiosyncrasy of the environment (such as solving conflicts or allowing for the creation of hierarchies in multi–user environments). Neither the language nor the underlying structure were designed as an end–user interface, but rather as underlying natural solutions over which user interfaces can be developed focusing just on interaction principles. Thus, the language can be seen as a *lingua franca*, capturing the essence of end–user

programming in a basic form of speech. Each form of interaction places specific requirements on speech while the programming essence corresponds to the way end–users think about programming and, thus, should be the same for every form of interaction. This work presents a solution to this underlying programming language and structure. According to the twofold strategy (i.e. language and structure), the most important contributions are summarized as follows:

1.3.1 Rule–based Language

This work presents a rule–based language for programming indirect control applications. Pointed out by Myers as those naturally employed by users in solving problems [93], rule–based languages preserve the natural essence of context–aware applications: *"When something happens, if some facts are present, then do something.* Thus, users' preferences are codified in this language as reaction rules in the form of Event Condition Action (ECA) rules. The ECA structure is used firstly as an application–independent mechanism of expression (since it encapsulates a method of commanding, not what to command) whose degree of complexity relies on the complexity of its atoms (events, conditions and actions) and can, thus, be adjusted to the users' skills. Secondly, it is used as an explanation mechanism since the very rule that produced an action serves as a valid human explanation to answer why that action was commanded. In addition, by tagging the actions of the ECA rules with a "confidence factor" adjusted by observing the reactions of the user to the action, this work opens the door to automatic learning of ECA rules.

In order to deal with Papert's ideal of "low threshold, no ceiling", the rule–based language is designed as a **base language of extreme simplicity** that can be extended —as the user's skills are developed— with more powerful structures. These structures, mainly *Wildcards* (designed to deal with generic concepts such as "any light") and *Timers* (designed to deal with time concepts such as "in the next 5 minutes") are designed in such a way that they are constructed using the same concepts of the base language, so each forward step in the learning curve brings the user closer to the next. Thus, **complexity is isolated in independent structures using the same base language concepts**.

As concerns the expression capabilities, this work presents a new approach to event composition and consumption policies. Thus, through the use of Timers and using an event logic instead of an event algebra, it **allows for defining context–dependent composite events** as well as **expressing mixed consumption policies**

1.3.2 Programming Structure

In order to deal with the multiple sets of preferences of a user or group of users, **we have distributed rules along many independent reasoning engines called agents**. Each agent has its own set of rules and is related to the user or group of users that created it, as well as to the elements they affect with their rules. Additionally, they can also be related to the environment they act on and can be tagged with the purpose they were built for. These agents are represented in the Blackboard (the abstraction layer) as another part of the context so they can be accessed and commanded like any other element of the environment. In doing so, we have created a simple structure for modifying the agents' status through ECA rules, allowing the users to create their own hierarchies by activating/deactivating agents according to context.

In summary, users can distribute their rules among different agents, grouping those they consider in the same conceptual structure. The grouping reasons are up to end–users and, in that manner, will be those directly matching their mental responsibility structures, not ours. By not forcing any bundle (such as activity or location) but allowing any, **modularization is done according to the end–users' mental plan** and helps them in locating responsibilities. Tagging each agent with its owner, affected elements and other optional categorizations such as location or purpose, together with the possibility of creating rules affecting agents (or using them as part of the context) **allows end–users to transfer their natural hierarchies to the Ubiquitous Computing domain** in a straightforward manner. In order to increase the potential of hierarchy construction, this work presents a multi–layer flexible mechanism that helps users (without constraining them) to easily create and combine different hierarchical domains.

1.4 Scope

Once we have defined end–user programming and indirect control as the main frame of this thesis, and before delving into more complex descriptions, we want to briefly describe what this system is intended for and what it is not, for the sake of a better understanding of the following pages.

This work focuses on the application–dependent problem of personalization and the inverse relationship between flexibility and simplicity noted above. Along these lines, it seeks an end–user centered indirect control mechanism for providing the user with the power to control everything naturally and easily. This work does not aim or pretend to develop any particular interface but, conversely, to create a programming mechanism that can be used through many interfaces. This preserves the programming structure despite the interface in use — making programming also UI–independent. In addition, it allows UI programmers to

focus just on UI issues. The idea behind this decision, as we will expose along this thesis, is that while programming (especially for end–users) is a mental static process, interfacing is a sensory/interaction process dependent on the context. Thus, UIs will be chosen just in terms of interaction capabilities: i.e. to program a reminder for a friend's birthday, a different UI would be preferred than if driving a car, taking a shower or working on a PC, but the underlying programming mental plan will remain the same despite the choice in UI.

This work aims to extend end–users' control over their environments by addressing the application–dependent problem and minimizing the inverse relationship between flexibility–simplicity posed above. That is, by providing a natural way for indirect control (i.e. everything is controlled in the same way). This is done in two ways: by increasing their control possibilities and by simplifying control complexity.

In order to define the goal (and, consequently, the scope) of this thesis, we must define the kind of control we are trying to provide. Thus, we will classify the different types of control according to two questions: "How is the environment controlled?" and "Who creates the control structures?". The first question differentiates between **direct control**, in which the user explicitly commands the actions to take place in the environment, and **indirect control**, in which the actions are not directly executed but codified as reactions to some context state triggering them. On the other hand, answering the second question,"Who creates the control structures?", we find two types of control structures: those designed and programmed by a **professional programmer** for the benefit of the user, and those programmed directly by the **end–user**.

This thesis focuses on indirect control mechanisms with a view to increasing end–users' control over their environments. Thus, it focuses on the **end–user programmed/indirect control** quadrant of Figure 1.5 with some incursions into the neighboring "end–user programmed/direct control" and "expert programmed/indirect control".

Even though the system was tested in a variety of scenarios, it is important to note that it was designed for **non–programmers**, with special emphasis on **personal environments**. This user–centered approach means that, while other systems pay special attention to problems such as the distribution of the computing load among network nodes or the minimization of bandwidth usage in a search of an overall computing efficiency, we have opted for a human approach, taking into account issues such as how social hierarchies can be exported easily to the automation processes, how the complexity of the system can be adapted to the user's learning curve or how the natural programming and managing structures of the end–users can be replicated in the Ubiquitous Computing domain, in order to avoid making them feel unsafe or alienated in their own environments. In addition to personal environments, the system was evaluated in educational, security

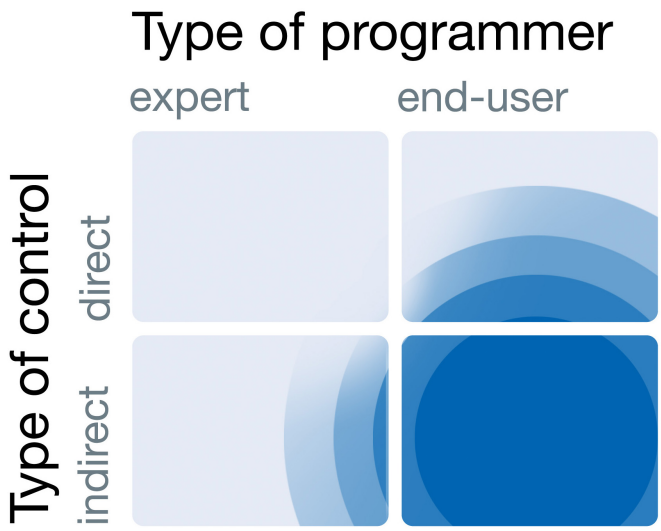


Figure 1.5: Classification of control interfaces according to the type of control it provides. It distinguishes between direct control, in which the orders are explicitly commanded by the user (e.g. turning on a light when a button is pressed) and indirect control, in which some context implicitly commands the orders (e.g. turning off the lights when nobody is left in the room). In relation to the programmer of the interfaces, it distinguishes between interfaces programmed by experts, in which a professional programmer decides what the user interface does, and end–user generated interfaces, in which end–users design their own control interface. Blue regions denote the focus of this work.

and research scenarios to test and enhance its expression capabilities. However, we must stress that the main target of this work is personal environments.

Finally, as stated before, this work focuses on capturing the programming essence of end–users in a programming language and structure. None of them are designed to be end–user interfaces, but to establish the common ground over which UIs are designed. Thus, UI designers can focus just on interaction issues while the programming paradigms are kept unchanged for end–users, despite the UI interface used in the interaction.

The scope of this thesis must be kept in mind in order to understand its essence in the wider perspective. We hope this will be easier after analyzing the most important examples of **"programmable environments"** in the following section.

1.5 Thesis structure

This thesis is structured as follows: Chapter 2 analyzes the state of the art of reactive Intelligent Environments. In Chapter 3, indirect control in Intelligent

Environments is analyzed, pointing out the most important characteristics of these environments, the requirements stemming from them and the solutions this work presents to deal with them. Chapter 4 describes the ECA–rule based language used as the kernel language of the programming system, while Chapter 5 explains the multi–agent structure used to modularize rules, the Agent's anatomy, execution model and a mechanism to manage hierarchies in Intelligent Environments. Chapter 6 illustrates the practical use of this system, in the different environments in which it was deployed, through a set of demonstrators, and user interfaces created to test it. Finally, Chapter 7 summarizes the most important conclusions derived from this work, presents some interesting lines of future work and reports its dissemination in and contributions to the research community.

Chapter 2

State of the Art

Ambient Intelligence (AmI) is a field of research posing many challenges, from integrating and communicating many different technologies at the lowest level to considering the psychological, social and interaction implications of a human population at the highest. That is, from the lowest hardware to the end–user. All these challenges stem from the same objective: to give the environment some intelligent capabilities. While some research has focused on how intelligently the environment perceives (e.g. activity or location recognition), most efforts are devoted to intelligently acting (for which intelligent perception is often needed). This is normally referred to as context–aware environments, in the sense that they act in consideration of what is happening in them. They have the ability to perceive what is happening and the intelligence to act accordingly.

This aim is translated into many research trends that can be classified according to their underlying philosophy regarding *where is the intelligence?*. Thus, while some trends aim to build intelligent environments, able to perceive and autonomously act in accordance to what they observe, certain others aim to provide the necessary tools for humans to build intelligent applications. But this classification can lead to misunderstandings: since any system is developed by humans, where is the line between an intelligent environment and a developer who has the tools to build it? Thus, we can look at it as a continuous back and forth from closed systems to open ones, depending on whom and how easily the context–aware capabilities can be extended or changed. In this sense, purely intelligent environments are in the closed end, in which Intelligence is in the environment (programmed once by someone) and any further improvement to the environment's behaviors will be done by the environment itself. In the middle there are those trends striving to provide some sort of tool for programmers to develop new context–aware applications. Application Programming Interfaces (APIs), Toolkits, languages or Frameworks are provided to ease the developer's tasks by

removing low–level information, homogenizing interaction or automatically distributing programs among nodes. At the other end of the classification are those trends aiming to put the end–user in the loop. Thus, this classification is one of continuous control, from system–centered, through programmer–centered to end–user–centered control.

Another important characteristic of AmI systems is their opacity. That is, the ease with which the whys of the environment's behaviors can be understood. Ranging from *blackbox* systems, in which the reasoning internals are completely hidden from human understanding (the best example would be the use of artificial neural networks) to *whitebox* systems, in which the reasoning internals are shown in a human–readable way and tools are provided to scrutinize them.

The implications of this double classification are best understood when analyzing the most relevant home automation systems. Since this work is focused on leveraging the end–user's control over the environment, we will pay special attention to whitebox end–user oriented systems.

2.1 The System in control

System–centered control systems are normally based on the assumption that the end–user is either incapable or unwilling to pay the cost associated with programming. Most of these systems are implemented by narrowing the automation domain, so the intelligence of the environment can be better tuned to that particular domain of automation and, thus, achieve a higher degree of competence.

2.1.1 Blackbox systems

Since every system–centered control system uses some kind of Artificial Intelligence technique, most are based on non–human–readable paradigms, such as Artificial Neural Networks, Hidden Markov Models or Bayesian networks. Therefore, most system–centered control systems can be considered blackbox systems.

The Neural Network house

Probably the most significant example of system–centered control is Mozer's Neural Network house (NNH) [91]. Taking advantage of artificial neural networks (ANN) and with the underlying philosophy that users do not want to program even a simple VCR, this project has two different goals: to anticipate the user's needs and to minimize energy consumption. This is done in a four–field domain (light, heating, water and ventilation) using the *discomfort cost* (a U.S. dollar measure for dissatisfaction) and the *energy cost* to balance both goals.

The NNH seeks intelligence in three different domains: context generation, goal definition and goal execution. Each is based on the previous one in order

to enable the house to "program itself" for the user's benefit. This is done by monitoring the environment, observing the actions taken by occupants and attempting to infer patterns in the environment that predict these actions. Thus, as in any other AmI system, the architecture is based on a context layer, in this case relying on the environmental state and occupancy models and using ANN to enlarge the context model with predictions about future actions (such us turning on the shower in 30 minutes). This context information is then passed to *setpoint generators*, in charge of determining the optimal value for each environmental variable (e.g. light or temperature) [90]. That is, what has to be done in general terms (e.g. increase the light to X). Finally, the setpoint profile is passed to the *device regulators*, which are in charge of translating the global goal of the setpoint generator into the minimum set of transformations to achieve it (e.g. increase $lamp1$'s intensity 2 points and lower $lamp2$ and $lamp3$ 3 points each). Setpoint generators and device regulators use dynamic programming and reinforcement learning.

Therefore, in the NNH, both establishing the overall goals and deciding how they are accomplished are system decisions. In addition, the system cannot explain its reasoning process in an understandable manner. It acts on the basis of what happened, according to what it sees happening and believes is going to happen. Thus, the user is completely out of the loop.

UMASS Intelligent Home

Another system–centered control approach can be found in the Intelligent Home Project [77]. This project assumes the existence of context retrieval technologies for identification and tracking of humans in the environment, identification of preferences profiles (including deadlines for activities) and assimilating the occupants' preferences for parameters such as temperature. Thus, the way in which preferences are obtained will determine how out of the loop the user is. Anyhow, as Mozera's et al. control was effected on an environmental variable basis, this project focuses on appliances. Thus, the UMASS simulated environment is controlled by intelligent agents that are associated with particular appliances. Each agent has associated tasks and negotiates with other agents over the resources when they are insufficient to meet the demands. Conflict resolution is done using TÆms [31] domain independent task modeling framework that describes primitive actions statistically via discrete probability distributions in terms of *quality*, *cost* and *duration*, and tries to select the course of action that best meets the current constraints and environmental conditions.

While probability distributions are easier to understand than weights in artificial neural connections, they are still far from being human–readable. In addition, even though new agents can be created for new appliances (so the domain can be expanded by developers) and the system takes into account the user's preferences

(and therefore may not be completely out of the loop), choosing the task to be done is still under the System's control.

2.1.2 Whitebox systems

While most system–centered control systems can be considered to be blackbox systems, some of them can be said to be whitebox, either because their knowledge representation is close to being human–readable or because they have been designed to be understandable by end–users.

MavHome

Regardless of who has control over the environment, the end–user's preferences must be taken into account. While UMASS assumes the existence of a technology to retrieve them, MavHome —closer to NNH— assumes that *people are creatures of habit* [124] and will provide some periodicity or frequency for a number of activities they perform in an environment. While this assumption opens the door to automatic learning through observation, the domain must be constrained in some way to make it feasible. In this sense, while NNH restricts the automation to just certain environmental variables, MavHome assumes that there is only one inhabitant in the home.

Observations of inhabitant behavior are encapsulated in even–based chains of a Hierarchical Hidden Markov Model (HHMM) in which actions and rewards are tied to the transitions between states in what is called a Hierarchical Partially Observable Markov Decision Process (HPOMDP) [37][115]. In addition to the HPOMDP, the observed data are also used to train a *prediction algorithm* and an *episode membership algorithm*. These algorithms are used by a decision–maker process to try to locate where in the HPOMDP model the inhabitant's activities are currently engaged. Once successful, the decision–maker looks ahead and decides on an action, if one exists.

Two of the problems of learning from what users do are that users cannot do all they would like to automate (e.g. turn all the lights at the same time), nor do they want to automate everything they do (e.g. watering the plants). In order to try to solve this problem, MavHome presents a user–centered subsystem, employing a rules engine for maintaining knowledge of user preferences, safety and security rules and constraints (such as specifying not to automate a particular item).

While this rules engine can be seen as a change to the underlying philosophy, MavHome uses it as another input mechanism. Thus, the rules are used as a feedback mechanism to train the system (trying to learn what to do instead of reading what to do). That is putting the control back in the system. Additionally, even though HMM may have an interpretable graphical representation, being

thus interpretable, they are not strictly readable. Thus, while presented here as a whitebox system, MavHome can be considered to be somewhere between blackbox and whitebox systems, choosing one or the other according to whichever system it is compared.

PRIMA

Another interesting approach to System–centered control is PRIMA [15]. While its goal is to address the problem of supervised learning in intelligent environments, PRIMA defines two qualities that a machine learning Intelligent Environment should have: an *understandable representation and reasoning* and *supervisor corrections* (feedback). For the latter, further on, three types of feedback are distinguished: action corrections, deletions and preservation.

To achieve this, the PRIMA context model consists of situations, roles played by entities and relationships between entities. It is a non–Bayesian model inspired by concepts in planning and knowledge representation used in robotics [26]. The situation is the cornerstone of the model, representing a particular state. Roles are assigned using acceptance tests comparing some entity's property with predefined values (e.g. a person is playing the "Lecturer" role if she stands next to the presentation screen). Entities, on the other hand, are defined as predicate functions on several entities playing roles (e.g. the identity relation may be created by comparing the names of two entities). A situation changes when there is a change in the role of an entity or in the relation between two entities. Thus, a network can be constructed using the situations as nodes and the changes in roles and relationships as the arcs connecting different situations.

System behaviors are directly associated with the situations of the network; therefore, since there is only one situation active at any moment, feedback is a straightforward process. An interesting feature of this system is what is called *situation splitting* and corresponds to different feedbacks for a single situation. Thus, while the supervisor perceives two situations (one for which she gives a positive feedback and another with a negative one), the system only perceives one. The situating splitting is conducted as a classification process, taking into account all the roles and relations of the training examples that were not considered relevant to defining the actual situation. While this is highly useful for automatically adapting predefined rules, allowing the system to be fine tuned to the user's desires, we believe that a context model based on roles and relations is oversimplified and may lead to situations in which the relevant element for splitting is left behind.

□

In conclusion, system–centered control systems aim to provide the environment with the necessary skills to infer what users want and do it for them. This kind of approach presents two main drawbacks: understanding users' preferences

and dealing with "noisy" automation domains. Understanding the user's preferences just from observation has the problems stated by Youngblood et al. [124]: users not being able to do all they want to automate and not wanting to automate all they do. Leaving these constraints aside, the amount of variables to analyze in unconstrained domains can make the learning process very difficult, probably requiring more training examples than the user is willing to provide, meaning that pruning the variables to be analyzed becomes a must. In this sense, we believe that approaches restricting the automation domain to specific tasks (e.g. lighting or heating) such as the NNH may fit best in real spaces than those considering special context cases (e.g. only one inhabitant in the environment) such as MavHome.

On the other hand, even the simplest scenarios (such as automating the windows to control temperature) may depend on more variables than expected, leading automation to fail in many situations. As an example [66], opening a window to control temperature may not be the right thing to do if it is noisy outside, there is a strong smell in the street, someone is allergic to pollen and the pollen count is high, it is raining or the windows produce glare on the TV. Even though the system considers all these variables in making the right choice, Intille and Larson point out a fundamental problem: *the more complexity the algorithms consider when making decisions, the less transparent those decisions will be to the home owner.*

Finally, while this kind of "opaque intelligence" (that the user cannot understand, nor explicitly change) may be well suited for dealing with general out–of–sight problems such as a presence simulator for vacations or saving energy to heat water, having people willing to share their control over visible things with an autonomous, incomprehensible environment, even more in their homes is, at least, debatable.

2.2 The System under control

2.2.1 Centered on the programmer

In order to allow the environments to evolve as new ideas, technologies or automation domains mature, some systems have decided to give the control to the developer. That is, instead of providing a particular automation domain, they provide some high–level mechanism to easily program new applications in the environment. Since applications are not fixed in these kinds of systems but can be developed by different programmers, an a priori blackbox/whitebox classification is difficult and will depend on the particular applications that are programmed.

ParcTab

At Xerox Parc, one of the pioneering research centers in Ubiquitous Computing, they defined the PARCTAB system [121], a prototype developed to explore the impact and possibilities of mobile computation in an office environment. Originally, the system was based on three types of devices of different sizes: *tabs*, *pads* and *boards*. Over this system, different context–aware applications where programmed but, of special interest for this work are the *context–triggered actions* [109]: Active Badge [120] based "Watchdog" and tab–based "Contextual Reminders".

Context–triggered actions are simple IF-THEN rules used to specify how context–aware systems should adapt by encoding a context triggering an action. As an example, the watchdog program monitors Active Badge activity and executes Unix shell commands in response. A user configuration file (containing a description of Active Badge events and actions) is loaded on start–up. Entries of the configuration file, codifying the IF–THEN rule, are of the form:

```
badge location event-type action
```

where *badge* and *location* are strings matching the badge wearer and current location, *event–type* is a badge event type (i.e. arriving, departing, settled–in, missing, or attention) and *action* a Unix shell command. As an example, a rule for playing a rooster sound whenever anyone makes coffee would be encoded in the following way:

```
Coffee Kitchen arriving ‘‘play -v 50 \~/sounds/ready.au’’
```

Even though this system was one of the pioneers of Ubiquitous Computing, it already presented some very interesting capabilities and can be used to point out some limitations. First, despite being designed for use by programmers, it uses the natural IF–THEN structure for creating behaviors. In addition, the actions were end–user oriented; since most of its end–users were programmers, a Unix shell script seems adequate and understandable for all of them.

Secondly, it allowed different users to have different preferences over the same objects, each carrying their own preferences in personal servers, thus tackling one of the most fundamental problems of Intelligent Environment automation: their multiple population. On the other hand, the simplicity of the language did not have the necessary flexibility to address complex problems: its triggers were fixed to a badge, location and event type (one of each not combinable with other context information) and, secondly, even though the architecture allowed different preferences to coexist in the environment, no mechanism for coordinating conflicting preferences was supplied.

Ubiquitous computing environment Operating Systems

Some systems have decided to adapt the classical computing paradigm to the intelligent environment (a particular computing system). That is, by creating an Intelligent Environment Operating System (OS). This is the case of Gaia [108] and PlanB [7], each of which present a different approach. Thus, while Gaia provides an ad–hoc operating system, with numerous modules specially designed to deal with ubiquitous computing problems (such as a presence service or a context service), PlanB focuses on adapting a Unix–based operating system to the Ubiquitous Computing domain, being able to use all the already existing potential of said OS to create context–aware applications. In this case, in particular, proving the potential of simple Unix functions such as *grep* or *pipe* to create powerful applications when the underlying OS is file–based.

In this sense, while both approaches have many features in common, one focuses on designing an OS specially adapted to the Ubiquitous Computing domain while the other tries to maintain all the capabilities (and already known functionalities) of a previous OS, adapting it to the Ubiquitous Computing domain.

This kind of approach, while extremely powerful to enabling the evolution of the environment, has the same problem as most developer–centered control systems: if a control structure is not carefully designed and imposed on every programmer, the environment may end up populated by multiple applications, each of which (programmed by a different programmer) will have its own ways to control, adapt to and communicate with the end–user, leading to the application–dependent problem posed at the beginning of this work.

Gator Tech

As with most developer–centered control systems, the Gator Tech Smart House [63] aims to build an intelligent environment that is able to evolve as new technologies do. Thus, they offer a development environment with various tools to help create smart spaces. This development tool is designed for expert programmers and offers a *context builder* to visually associate behaviors with context in the form of a graph, a *service composer* to browse and discover services as well as compose and register new ones, a debugging tool and a simulator.

While this kind of approach allows the house to evolve and incorporate new domains of interaction, it leverages the control of developers without increasing the end–user's control. In addition, allowing developers to create their own applications (often needed) leads to a multi–author application populated environment in which the user has to deal with different control paradigms for each application (according to who its programmer was). That is, the application–dependent problem posed at the beginning of this work. This problem is common to many systems (such as Bardram's java context awareness framework [9] or Dey's et al. Toolkit [34] for the Aware Home [72]) providing high-level tools for creating ap-

plications such as toolkits, frameworks or APIs in which services are provided to communicate with the environment but the coding and resulting interfaces are up to the developers. This application–dependent problem can be seen, in the Gator Tech house, in the multitude of independent applications populating it: smart blinds, smart bed, smart closet, smart laundry, smart bathroom, smart mirror and so on.

ReBa

As part of the OXYGEN project at MIT, Kulkarny proposes a Reactive Behavioral system for the Intelligent Room [16] (ReBa). This system was born as an alternative to a previous reactive rule–based system (developed for the same project) codified in JESS [41], a java language based on *facts* and *rules* implementing the Rete algorithm [38].

The new system, ReBa [75], is based on associating reactive actions at the activity level. The base of the system is the *Behavior bundle*, a group of reactions associated with a particular activity. Behavior bundles are organized hierarchically in a dependence tree in which, for example, the *Brainstorming* bundle depends on the *Meeting* bundle, which in turn depends on the *Occupied* bundle. Therefore sub–contexts of a particular context are defined in the tree as different branches of the same node.

Through this tree, behavior agents (or bundles) form relationships with each other to prioritize themselves. This prioritization is translated in the running system into two different relations among bundles: *overriding* and *depending*. The depending relation means that a particular behavior bundle cannot be activated (even though the context matches) if the bundle on which it depends is not active. Thus, the *Meeting* bundle cannot activate if the *Occupied* bundle is not active. If an activity is performed within another, the bundle corresponding to the one started second activates on top of the first, *overriding* it. Since similar actions in different bundles are given the same name, one bundle can inform the other that one of its actions is suppressing one of the actions of the other. Behavioral agents are implemented as *Metaglue* [105] agents, a programming language for multiagent systems.

Kulkarny states that *making the design of new behaviors easy for all users is a daunting task. But making the design of new behaviors easy for those users comfortable with programming —say, other Intelligent Room developers— is much easier* [76]. We believe that behaviors associated with an activity level are too personal to be tackled by a professional developer. Personal environments being what Kidd et al. call *free–choice environments* [72], the amount of activities can be extremely large and dependent on the inhabitants of the environment. In addition, forcing the association of behaviors to activities may not be specially well suited to dealing with some preferences naturally associated with other domains,

such as the preferences of a particular person for a particular devices (despite the activity) or a general domain (e.g. energy saving).

Considering the lack of control a user may suffer in an environment programmed by third parties, ReBa provides a mechanism through which users can specify *macros*. While we believe this is a good step forward, user–defined macros are not integrated with the rest of the system but have to be invoked when desired. In addition if users find that the system is not performing as desired, they can deactivate it, leaving just the macros. This deactivation mechanism is a good feature of this system but since it requires choosing between "everything or nothing", a low performance of one bundle may lead to a loss of all the advantages of the rest.

RuleCaster

RuleCaster [12] is an interesting example of a developer–centered control system. While its main focus is on freeing programmers of the low–level details of a sensor network when writing applications (allowing them to build applications for the network as a whole rather than for each node of it), it presents some interesting features in relation with the programming language too.

Firstly, applications are expressed as rules, pointing out a requirement that will be central in end–user–centered control systems: applications are understood both by humans and computers. Each application is composed of several blocks of rules, each of which can specify *space*, *time* and *state* constraints to be satisfied in order to evaluate the rules inside the block, where the time specifies how the conditions of the rules have to be satisfied (e.g. simultaneously). Each rule is composed of a goal and some conditions. The goal can either be an action or a state change. If a state rule is satisfied, the new state is added to the current network state (associated to the *space* of the rule block) e.g. if a rule with goal "STATE hazard" is satisfied in a rule block with "SPACE(door)", all rule blocks with "STATE(door:hazard)" will be activated.

Even though this system is focused on automatically distributing programs along the nodes of a sensor network, it provides some good ideas in relation to programming languages, such as using a human–understandable language or associating rule blocks to particular domains (states in this case). Nevertheless, its rules are not provided with triggers, making them unsuitable to expressing some contexts. In addition, associating rule blocks just to a state makes it difficult sometimes to populate the environment with several behaviors codifying different preferences for the same domain.

SmartOffice

The SmartOffice [45] presents a framework for coordinating and programming the different modules of an Intelligent environment. Modules communicate (using an

XML–based protocol) with a supervisor acting as a resource server. The supervisor is programmed using a rule–based language with two types of rules: backward and forward. Backward rules are programmed in PROLOG while forward rules are programmed in Clips.

Not all the agents should be aware of the resources given by other modules; thus, instead of asking other modules for their particular services, they can ask the resource manager for particular information (e.g. user's location). This approach is similar to that of other blackboard-based systems such as Haya's Blackboard [59], in which information is stored in a common repository so applications can make use of it without knowing who generated it or how.

One of the characteristic features of this system is that each agent manages a specific room function, following Coen's intent [22], pushing messages in relation with their specific function. Thus, a "gesture–detector" agent may have a rule that, on an "ON-gesture" recognition, it pushes a "light–command" message. The "light" agent may have another rule that, on "light–command" messages, it pushes a "light" message with an ON value. In this manner, push messages are propagated through the agent network, using the resource manager to hide the details of the generating module and low–level reasoning. Backward rules, on the other hand, are used to propagate the pull messages through which different modules ask for contextual information.

While this system presents several advantages, context generation and application programming are treated at the same level (the former as backward rules, the latter as forward rules). This approach leads to a system split into modules (each of which has some context and some "code") instead of into process layers in which information and applications are clearly separated. In addition, forcing each module to manage specific functions eases system communications, but forces programmers to split their application into the different modules it uses. In other words, while programmers may have a global idea of their programs, they have to split it and distribute it along the network in order to program it (this is one of the problems tackled by RuleCaster [12]). While this may have its drawbacks for professional programmers, it is unacceptable for end–user programming.

Implicit Human–Computer Interaction

Another approach to developer–centered control is that of A. Schmidt [110], looking for a way to specify what he called Implicit Human–Computer Interaction (IHCI). That is, context–aware reactions. This was done in the form of a rule with a context and an associated action codified in an XML–based language.

This language was designed to introduce the concept of IHCI and meet the requirements of certain projects. Thus, it has some flaws such as, for example, conditions must consist of a variable and a value. This means that conditions

cannot be built to compare the values of two variables or to express generality (e.g. "when *some* variable..."). Nevertheless, some of the characteristics of the language have to be considered. Firstly, the context was defined as a group of variables (e.g. pilot.on or sensor_module.touch) wrapped in a context that could be defined as "one", "all", "none". Thus, through the "none" wrapper, contexts could be defined as a negative statement. In addition, of special interest is the way in which actions are associated to contexts. Contrary to most systems, the context–action relation is not fixed in IHCI, but can be triggered when the context is detected, when it is no longer valid or while it is active with the *ENTER*, *LEAVES* and *WHILE* relations. We believe that these kinds of relations should be considered when designing a control system in order to measure its expressiveness.

Active Database Management Systems

Some researchers have decided to use an Active Database Management System (ADBMS) approach and apply it to Intelligent Environments. These provide interesting features regarding the potentials of the language, since they inherit many capabilities from the well-studied field of ADBMS. This is the case of Augusto and Nugent's Temporal Reasoning Language [6]. This language is based on Event Action Condition (ECA) rules. One of the greatest potentials (in our point of view) is that the language is founded on Galton's proposal to represent events and states, inspired by natural language-based research [46], which brings it closer to an end–user centered control system.

The language relies on three blocks, *ON*, *IF* and *THEN*, and a series of predicates. The most important are *Occur* and *Holds* for events and conditions, respectively. In addition, the language is enriched with time reasoning, considering both *intervals* and *instants* and events (E) and states (S). Therefore, $Occur(E, i_1][i_2)$ denotes the occurrence of event E in the instantaneous time reference represented as $i_1][i_2$ were $i_2 = i_1 + 1$. On the other hand, $Holds(S, [i_1, i_2])$ denotes that the state S holds in the durative time reference $[i_1, i_2]$ were $i_2 > i_1$.

This language presents some very interesting features, temporal algebra operators, complex events and negation (treated as negation as a failure), among others. These features served as an inspiration for this work and are discussed in more detail in Section 4.1.4. Conversely, regarding the language, while its power to describe most scenarios cannot be denied, temporal concepts are unavoidable in describing anything. Everything must be described in terms of temporal relations making the use of this language in an end–user centered control system (according to a study conducted by Dey et al. [33]) more difficult . Dey et al. observed that 56.4% of all the rules involved objects or the state of objects, 19.1% activities, 12.8% locations and only 7.6% time. Thus, to deal comfortably with 7.6% of the tasks, this language adds the complexity of dealing with time to all

of them.

CONON

Regarding the use of rule-based languages for expressing desires in the form of reactions to context states, some projects have put this mechanism to other uses. This is the case of CONON [119], an OWL-encoded context ontology. CONON studies the use of logical reasoning to check consistency of context information and infer high–level context from low–level context. To do so it uses ontology reasoning rules, defined in OWL, to describe properties such as Transitive or inverseOf. Thus, if the property location were defined in the ontology as Transitive, knowing that Wang is located in the kitchen and the kitchen is located in the first floor, the system would reason that Wang is also located on the first floor; therefore the consistency of context information is acquired automatically. Additionally, in order to provide a mechanism for extracting high-level context from low–level information, the system allows for the creation of User–defined context reasoning rules in a manner that can define a rule of the type: "If Wang is located in the bedroom and the bedroom light level is low and the bedroom drapes are closed then Wang is sleeping". We believe that this property is also desirable in end–user–centered control systems so end–users can not only program their environments but also enhance them with new context information with which to work.

To achieve this, CONON relies on the importance of a descriptive event algebra, adding to the traditional set of operators (AND, OR, ANY) an extra one to compose more complex events (sequences, aperiodic and periodic) [113].

In the same line of work, the OCP system, developed at the University of Murcia [98], involves "middleware which provides support for management of contextual information and merging of information from different sources". Based on Semantic Web technologies, a context inference mechanism was developed based on rules such as do–if rules or do–for–all rules. The inference is carried out using the SWRL guidelines [3] and the Jena platform as the inference engine.

2.2.2 Centered on the end–user

Both system–centered and developer–centered applications are normally characterized by having the end–user "out of the loop", in the sense that a third party decides the choice of which domains are automated or what the relevant factors for automating them are. In order to leverage the control end–users have over their environments, end–user centered control systems provide alternatives for end–users to program their environments, choose what domains they want to automate, how they want to do it or simply take or leave the system's advice. Either way, those systems are characterized by undoubtedly having the human

"in the loop"

Blackbox systems

While most end–user centered control systems are whitebox, there are some inspiring examples of blackbox solutions.

Changing Places/House_n

The first example of blackbox end–user centered systems is a great example of the shift in philosophies between system–centered and end–user centered control systems (even when the technological approach may be quite similar): in Intille's and Larson's own words, *the shift from the "controlling" home to the home that is supportive* [66].

House_n is a project for creating supportive technologies that help people to create and customize environments, live long and healthy lives and integrate learning into their everyday activity in the home. Thus, their vision is not that of system–centered control systems in which the computer technology is "ubiquitously and proactively" managing the details of the home but, instead, they look for a system able to present information to the user at the right time and place so that she can learn and use it to improve her life. That is, empower people with information that helps them make decisions.

As an example of this implementation, they have built a PDA–based prototype to help users prevent congestive heart failure (CHF). The system uses a Bayesian framework to integrate evidence of CHF, as well as to choose meaningful questions to ask users in particular contexts. When users pull out their PDAs, a simple question is prompted and, according to Intille and Larson, quickly answered by users with almost no interruption since, they argue, this is a "point of behavior" in which the user *has already made a decision to stop whatever he or she was doing* [66]. This information is added to the preventive diagnosis profile and when a progression towards a CHF is found, the person is notified.

We believe that this kind of approach, in which the system is only used to inform the user, may be quite useful and can indeed help people to be aware of their environments and be more sensitive about the effects of their actions. The example of displaying information on the refrigerator as somebody approaches it (comparing the average time it is left open with the average time of the 10 closest neighbors [65]) is quite inspiring in this sense.

On the other hand, users may want to automate certain tasks and, no matter how much someone wants to teach them, showing information about how they must do them will hardly satisfy many expectations. In addition, any communication between the system and the user is a potential interruption (e.g. Microsoft Paperclip) and, even choosing "points of behavior" like House_n, may become an

unwanted and intrusive practice. Finally, when recommendations are not clear to the user, a blackbox reasoning process such as a Bayesian network does not allow the user to scrutinize the system and understand why the recommendations were made.

A CAPpella

Other examples of blackbox end–user–centered systems are those of programming by example. While the end–user is the one in charge of giving the example, understanding the particular action of the environment requires understanding the internal processes through which the system considered the particular situation to be another instance of the former example.

This is the case of a CAPpella [32], a programming by demonstration system (PBD) that uses the Dynamic Bayesian Network framework equivalent of a Hidden Markov Model to support activity recognition. The key idea of a CAPpella is that it empowers recognition by utilizing the users' natural abilities to understand their own behaviors. As stated in [32], while defining a meeting may be a daunting task for a user (and definitions among users may be different), recognizing a meeting when seeing one is a straightforward process.

In this sense, when the user wants to create a context–aware behavior, she starts the a CAPpella recording system (software running on a machine). While a CAPpella is running it captures data form all the sensors that are available to the system. When the user is finished, she stops a CAPpella recording and trains it on the recorded data. To do this, a CAPpella shows the user the events detected and a video of the recorded period. The user can select the streams of information she considers relevant. When the user repeats this process a number of times (over a period of days or weeks), the ability of a CAPpella to recognize the behavior with new data is improved.

Allowing the user to select the relevant events from the recording and using them to train improves the accuracy of the system and reduces the number of training examples needed. This work can be tedious, however, and has to be repeated a number of times before the system learns. In addition, while this approach allows users to specify which things should be automated (and which not), it does not have a solution for the other problem of trained–by–observation systems: users cannot physically do all they may want to automate.

Whitebox systems

Finally, most end–user centered control systems can be considered to be whitebox systems. In this sense, the vast majority of them are focused on end–user programming. To achieve this, some of them present particular UIs while others focus more on a language. As we will see, one of the characteristic features of

such systems is the domain they are intended for: while some of them restrict the programming domain to a particular space or task, others have been designed to deal with different domains of automation. The simplicity and flexibility of the programming solutions reflects whether they focused exclusively on a novice population, an experienced one, or whether they tried to deal with both.

Intelligent Office System

Following a path somewhat close to that of the House_n project, the Intelligent Office System [20] explores the *tension between user control and proactive services*, paying special attention (contrary to the House_n project) to supporting user comprehensibility of system behaviors. Thus, one of its main requirements is *scrutability*, in some way inspired by McFarlane and Latorella's work [71].

The system is based primarily on a context–retrieval process (sometimes gathered automatically, sometimes by hand, through a UI, by the end–user e.g. "I'm in my office"). Once a context history has been retrieved, a set of rules is induced which represent the user's preferences. Based on these rules the system provides suggestions to the user when the environmental context changes (e.g. "shall I turn the fan off?"). Suggestions can be accepted or rejected by the user in a UI that also serves as a direct control mechanism for acting remotely on devices.

In order to provided the desired degree of scrutability, the Intelligent Office System uses fuzzy decision trees [67] for inference. The rationale is that *the rules governing the system's proactive behavior can be expressed symbolically* which, in turn, enables users' understanding of these rules. In this sense, this system is limited to a small domain of environmental variables (i.e. temperature, noise level, light and humidity) and devices (i.e. window fan and heater). Continuous changes in the environmental variables are mapped onto a fuzzy discrete category (e.g. while 14° is 1.0 cold, 20° is 0.5 cold, 0.5 mild). Since the states of the different devices are associated (through observation of context–history) with the continuous values of the variables (e.g. 23°, 55 noise level, 30 humidity and 52 light is associated with window closed, fan off and heater off), they can be associated with the fuzzy discrete values of the variables (e.g. there is a 0.6 probability that the window is closed, the fan is off and the heater is off). Based on this information, the system prompts a suggestion when the probability of being in a state exceeds a *proactive threshold* and some device is not in the state it should be. The control GUI, besides prompting suggestions and allowing the user to directly control the devices, allows the user to change the proactive threshold, the boundaries used to separate different categories (e.g. mild from hot), see the rules generated by the system and add her own rules. End–user generated rules are always of the type "When temperature = V AND Noise Level = W AND Humidity = X and Light = Y and Window = Z THEN *action*", where V,W,X,Y and Z are either the discrete values of the variables (e.g. cold, mild

and hot for the temperature) or the value *any* and *action* altering the state of one of the devices.

In conclusion, the Intelligent Office System provides an interesting example of balance between system–centered and end–user–centered control in which the end–user is always on top of the command chain. Nevertheless, it offers a very restricted domain of automation and language expressivity for end–user generated rules.

Alfred

Alfred [44] is a natural end-user programming interface for Intelligent Environments developed at MIT. This system is part of the Intelligent Room Project [57] and designed as the heir to Rascal [43] and ReBa [76]. It is intended to allow developers to deal with adaptive and reactive components in Intelligent Environments.

Alfred is designed as a multi-modal macro recorder specially designed to be programmed through natural language [21]. Through Alfred, users can verbally ask the system to record a new macro. Macros are primarily spoken commands. An example would be *Turn on the main lights. Open the drapes. Turn on my desk lamp. Say "good morning." Stop recording.*. After a macro is recorded the user assigns a name to it (e.g. *Good morning, computer*). Every time the user names the macro, the system executes it. In addition, users can add some hardware triggers to the macro or call other macros from within it (i.e. *When I press this button [user presses a button] run the "Good morning, computer" sequence*). Despite all this, macros are simple task sequences lacking explicit conditionals.

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

e-Gadgets

In the e-Gadgets project and over the Gadgetware Architectural Style (GAS) [70], Mavrommati et al. [81] have designed a mechanism through which to "program" in-home devices. GAS includes a *plug–synapse* model describing objects as entities whose abilities can be inter–associated. Thus, people can associate compatible plugs of different entities to create a composition of the respective services and functions (i.e. synapses). These synapses can be visualized by the user, allowing for a better understanding of the Intelligent Environment's insights in order to deal with one of the requirements of the system *"The application behavior should not surprise the user.* This requirement is common to many other systems described in this Chapter and points to the necessity of debugging tools, as well as of mechanisms to make the users feel in control of their environment. As an

example to better understand the system, the condition "the book is on the desk" will be coded by plugging the *T–plug* of the book entity to the *proximity* ability of the desk entity.

While it poses an interesting programming paradigm, we believe that connecting elements is not a natural programming method (see Section 3.3.2). Additionally, this work is based on the concept of object as the main programming block. While we completely agree with this, as we explain in Section 3.2.2, we believe that some other concepts such as activity or time may be used for programming too. Basing the programming language on connecting the abilities of different objects has forced e–Gadgets to model some activities as abilities of the objects. We believe that this approach may be misleading in an end–user programming scenario in which the user cannot remember which object holds which activity. This problem is subsumed by e–Gadgets by providing a transparent view of each object's capabilities, so the user can scrutinize the system and look for what they need. Still, we believe that making a model closer to the end–user view will ease this process.

In addition, conditions may not only depend on the relation between two entities, but it may also depend on many other factors, such as time or the relations between relations or properties (e.g. "the locations of Manuel and Pablo are different", "a light is brighter than another" or "a person who knows Pablo but not Manuel"). Finally, in this approach, the configuration of the system is treated as a whole, making it difficult to coordinate preferences from different users.

MediaCubes

In the University of Cambridge's AutoHAN project [14], a range of programming languages for home automation have been explored. Firstly, Iota.HAN [11], a concurrent XML scripting language, was designed as a domain–specific language to express device behavior within the context of Home Area Networking. Being designed as a system programming language, aimed at experienced programmers, it corresponds to the developer–centered control of our classification.

Within the same project, *Lingua Franca* [54] was developed as a framework facilitating the use of multiple scripting languages. Taking the idea of Iota.HAN and trying to tackle its unsuitability for end–users, they developed Mediacubes, a TUI similar to the mediaBlocks [118] (a series of cubes used to move media from one place to another) but with the major difference that they provide a programming interface, rather than an interface to a particular functionality. More than an interface, the MediaCubes can be considered a language supported by Lingua Franca. Thus, *programs are not constructed by using words or graphics, but by manipulating physical objects* [54]. One of the problems of such a language is that it is restricted to a particular UI, in this case the cubes, that may not be

well suited for all kind of situations. In this sense, Lingua Franca supports two other languages that can be translated and used indistinctly with MediaCubes, a script language and a graph–based GUI language, but they are not as well suited for end–users as the MediaCubes.

In order to build a program, cubes are combined dynamically. Once two faces touch, the cubes may be separated without removing the association. This, while allowing the physical barriers inherent to the model (such as the physical impossibility of making three faces of three cubes touch each other) to be broken, may make the process more complicated to the end–user since we can say it is not a WYSIWYG (What you see is what you get) UI and a cube may be connected to another while you are seeing them in two different corners of the table. Besides the interface related problems, MediaCube is based on the script language of Lingua Franca and relies on a "Do–When" cube to specify causal relationships. This particular cube codifies a script within its four faces: Do, When, Whenever and Script. The script does what is connected to the Do face when (or whenever) whatever is connected to the When (Whenever) face happens. This script can be used as an event to connect other cubes by connecting its Script face to the Do, When or Whenever faces of other cubes. Thus, it is possible to nest scripts to compose more complex programs. The Script face of the outer program is then connected to a "Submit" cube in order to load the program into the Lingua Franca database. MediaCubes is a language for generating programs, but it cannot be used to check, modify or delete any already running programs.

As an example, let us consider a simple program to codify "When I go shopping, if I run out of fish, remind me to buy fish". To do this we need two Do–When cubes (A and B), and some other cubes for the events. The end–user should code this program by combining the cubes in the following manner (we use the notation X:Y–A:B to express "face Y of cube X is connected to face B of cube A"): *A:Whenever–C:OutOf/fish A:Do–B:Script(B:When GoShopping B:Do Order/–)*.

While this project provides a good source of inspiration, such as having a base language that can be generated from other places (other languages in this case) or providing an end–user interface to program the environment, we believe firstly, that constraining the language to a physical interface, while well suited for particular scenarios, will fail in many cases (in which the physical objects are not at hand, or the amount of different domain elements is too large, for example). Secondly, that trying to untie the TUI from its physical barriers to surmount some of its problems may be confusing to end–users, since it breaks their preconceptions about physical objects. Finally, and most importantly, while the interface may be well suited for end–users, a script piping language may not be natural to most of them and may be the true bottleneck of the system.

SiteView

Also exploring the possibilities of TUI, SiteView [10] offers a limited programming TUI that focuses on a small control domain. This interface was created with Papier–Mâche [73], a toolkit for rapidly creating tangible interfaces with computer vision and RFID. Nevertheless, while SiteView focuses on controlling a few elements (light and thermostat) in a single room, based on just three conditions (day, time of day and weather), it offers some interesting views of end–user–centered control.

Firstly, despite the strong limit on the automation domain, more than 40 tangible elements were created, revealing some limitations of an icon–based TUI approach. Some good ideas were also put forth. Primarily, SiteView provided an interface for accessing and modifying existing rules. This feedback was displayed to the user as an English–like sentence, supporting a more transparent user understanding of system behavior. Secondly, the UI provides an environment display that can be used to see the effect of a rule (i.e. How will this rule change the environment) as well as to check the effects of all rules in some conditions (e.g. How will the system react to a rainy Sunday morning). Thus, leaving aside the expression limitations of the system, SiteView provides good ideas for scrutability and testing, very important issues in inhabited end–user programmed spaces.

iCAP

Another example of an end–user–centered control system is iCAP [33], designed to allow end–users to visually design context–aware applications. iCAP does this by providing an UI to create if–then rules by dragging and dropping elements into a matrix in which the vertical dimension represents the AND operator and the horizontal dimension the OR operator (this is based on Pane and Myers' match form scheme [100]). Elements are provided by the system and can be created by the user through widgets by selecting a name, a category, an icon and a type (binary or 1–10 range). In addition, iCAP supports spatial relationships (e.g. "adjacent room") and temporal relationships (e.g. "5 minutes after I turn on the lights").

While this system presents many advantages, it was designed, like the MediaCubes, to support a particular interface that may or may not be suited to every possible scenario. In addition, while this interface is very well suited to novice users, it may not be so to more expert users. As of this writing no further steps have been taken to extend the language to other interfaces, assuming that the rules may be generated with other UI, more suited to expert users. The language has no events, so "When the TV is turned ON, if the light is ON, then turn OFF the light" will be expressed as "If the TV is turned ON and the light is ON, then turn OFF the light", leading to the unwanted reaction of turning OFF the light when the user turns it ON while watching TV (see Section 4.1). In addition,

it lacks powerful algebra to describe composite events or consumption policies so, even assuming a more powerful UI, the underlying language will still have limitations.

ACCORD

Accord [107] has developed a model in which every device in the environment is represented. While this is the first step towards perception, presented by most systems centered on developer control, Accord aims to allow the user to benefit from it too. To do so, it presents a Tangible User Interface (TUI) based on jigsaw-like pieces that can be piped to create control structures. This representation corresponds to their main requirement for simplicity. Thus, as they state, *"Our emphasis is on reconfiguration rather than programming and we do not seek the richness of programming expression allowed by iCAP"*.

Most of the jigsaw pieces represent particular devices, acting as events, inputs or outputs. As an example, piping the doorbell, the camera and the mobile phone jigsaw pieces will make the system send a picture of the entrance to the mobile phone when somebody rings the doorbell. Some other pieces codify high–level events or more complex actions. Thus, there is a jigsaw piece codifying a "grocery alarm" associated, possibly, with a barcode reader in the trash, and another piece for "adding to a list". Piping the grocery alarm to the adding to a list to the mobile phone jigsaw piece will make the system send a message to the phone with the shopping list every time a particular food runs out. In addition to the TUI, the jigsaw metaphor has been also implemented as a GUI.

While this is a clear example of end–user–centered control, it acquires the required degree of simplicity without considering the flexibility to deal with more complex scenarios. Thus, while probably well suited for novice users, it presents strong expressiveness limitations for adapting to the growing skills of more advanced end–users, breaking Csikszentmihalyi's [27] notion of flow of motivational factors in learning (see Section 3.2.2).

CAMP

CAMP [116], on the other hand, tries to balance simplicity and flexibility, allowing users to build more complex control structures to control their environments. Instead of an iconic UI like Accord's jigsaw, CAMP uses the Fridge Magnet Poetry metaphor, based on word tokens that can be arranged and combined to build a "poem", i.e. a sentence codifying a control statement. This metaphor is implemented as a GUI to assist users in searching for a word among the pieces. In addition, CAMP does not require end–users to use any particular structure of speech, allowing them instead to express their sentences in their own terms, only constrained by the existing pieces.

Once the end–user has composed a sentence, the system automatically translates it into instructions and parameters for devices, using a custom dictionary to reword, restructure, decompose or replicate the user's terms. This translation is prompted in the interface as feedback to the user, making the debugging process easy and natural. Thus, "Jim or Jane in the kitchen" is shown as "[Jim in the kitchen] and [Jane in the kitchen]".

CAMP is built atop the INCA infrastructure [117], an abstraction layer supporting interfaces for capturing and accessing information, components for storing information, a way to integrate streams of information and the removal of unwanted data. Thus, while the expression capabilities of CAMP are much wider than those of other systems, such as the jigsaw pieces of Accord, its domain is restricted to programming capture applications such as "Record picture in Billy's bedroom at night" or "When Jim, Jane and Billy talk, record and remember for 20 minutes".

This system has influenced the design of one of the UIs we developed and some of its characteristics will be described in Section 6.2.2.

2.3 Summary

In conclusion, we can see that two different philosophies drive most research into Intelligent Environments, one foreseeing a future in which the environment reminds an Artificial Intelligent entity, acting autonomously on behalf of the inhabitant, the other in which the environment enhances the control its inhabitants have over their surroundings. In addition, some middle–ground systems have decided, centering the control on developers, to restrict the autonomous intelligence of the environment, allowing it to evolve easily to adapt to new technologies, ideas and domains of adaptation, while others have chosen to restrict the control end–users have over their environments on behalf of more robust and complex applications.

In this sense, our philosophy can be described in Taylor's words:

> *An environment's smartness should not be measured by how much and what variety of data it can sense, or by the complexity of things it can do with that data. Rather, we might imagine an environment's intelligence to be about how clearly the environment's workings are revealed to us (without making the world more complex), and how easily such workings can be harnessed. The business of designing intelligence into an environment should thus be concerned with building things that we can get to work intelligently for us.* [114]

This philosophy drives us to look at end–user–centered whitebox systems as the solution for achieving our particular vision, but many of the contributions of

other systems may enhance our requirements.

Thus, when looking at end–user–centered control solutions as a whole, there are a number of characteristics that influence our work.

Firstly, while UI are a necessary tool for end–users to control their surroundings, we believe that the control system must not be driven by the UI interface, but by the internal reasoning structures of end–users. UIs will then be designed as an interaction mechanism, designed to suit the specific interaction needs of a particular task or scenario.

Secondly, while end–users need a simple programming language, increasing degrees of expertise and complex control domains will require a more advanced programming structure. Therefore, a base language suited for novice end–users should not restrict the capabilities of users with more expertise. Many developer–centered control approaches provide some good ideas about how to leverage the expression capabilities of a programming language. The challenge will be to balance this flexibility with the simplicity required to deal with end–users.

Finally, the automation domains may be wide and varied, as may the environment's inhabitants. Systems constraining their control domains (e.g. to a particular task, a number of simultaneous users or a particular bundle for preferences) fail at some point or another when facing an "in the wild" scenario.

These systems have laid the groundwork over which this thesis has grown. An analysis of their strengths and weakness, as well as the idea we have about a future in which Intelligent Environment technologies leverage people's control while preserving their social structures and freedom, have led to the requirements detailed in the following chapter.

Chapter 3

Indirect control in Ambient Intelligence environments

Ambient Intelligence (AmI) is a branch of research born from M. Weiser's 1991 vision of a world in which computers are seamlessly integrated into it, vanished in the background and interconnected [122]. It is based on environments rich in perception and interacting capabilities and characterized by the heterogeneity of environments, inhabitants, elements, capabilities and uses. Factories, personal environments, transport systems, hospitals, airports or educational environments are some examples of physical environments where AmI is applied. Energy saving [92], remote monitoring of relatives [94] or environmental control mechanisms [2][88][12] are some of the applications it studies. While many of them require some sort of automated reasoning, they have received much less attention than other domains such as business processing or the semantic web.

Sharing some characteristics with them, the idiosyncrasies of Personal AmI environments differ from those in some important points. Like business environments, they are physical but, contrary to them, they are highly personal, making benefit and reward harder to define. Like the semantic web, the amount of services and capabilities available can be extremely varied but of a different nature and bounded to specific constraints such as physicality or privacy.

When talking about Indirect control, we refer to mechanisms for actuating through implicit commands. Many forms of today's automation in buildings, such as turning lights on when somebody enters a room, belong to this type of control. Some AmI research trends have decided to narrow the domain of study to finding solutions for specific problems, such as saving energy, while others have focused on developing mechanisms to freely program the complete domain of the environment. Trying to solve the application–dependent problem (see

Chapter 1.2), this work focuses on the latter.

In this chapter we will analyze the special characteristics of indirect control in Ambient Intelligence (AmI) environments (see Table 3.1), the requirements we extracted from them (see Tables 3.2, 3.3 and 3.4)and the solutions we applied to deal with these requirements (see Table 3.5).

3.1 The nature of Ubiquitous Computing environments in relation to indirect control end–user programming

Intelligent Environments possess two characteristics that make them especially unique: their physical-computing nature and their human population. Each of them presents special features that have to be taken into account to design an indirect control end–user programming system.

3.1.1 Based on human factors

When talking about AmI, the most important issue of concern is that they are populated environments. Human–side characteristics arise from the way they interact with and perceive their environment. What people know, how they think, what they want or how they organize differs among groups (or persons), environments and situations. In addition, there are some social characteristics, strengths and limitations imposed by human psychology.

Computing technologies, when applied to personal environments, must deal in the most sensible way with human nature if they are to succeed.

Heterogeneity

First of all, people are different. Age, education, genetics and many other factors affect the way people think or reason, what they know and how skilled they are. Thus, when designing a system for end–user programming, we must remember that end–users will have **different degrees (or an absolute lack) of expertise (H1)**. In addition they will understand different things as problems. For example, forgetting a glass of water in the kitchen before going to bed may seem like a real problem to the elderly while it may go almost unnoticed for a 21-year-old girl. Furthermore, for the same problems, different people use different solutions, adapted to their way of thinking or to what they want to do. Thus, under a shared problem as may be that of watering plants, some persons may prefer to be reminded when to water them while others will prefer to automate the watering process. Thus we find in human populations a **heterogeneity of goals and means** among people **(H2)**.

Preferences

This last problem is strictly related to the heterogeneity of preferences among users, of special importance for this work.

Computing technologies have been applied to many human-populated places but personal environments present a main difference over all of them: they are personal. When talking about hospitals or factories, the inhabitants participate toward achieving an external goal — i.e. to reduce treatment time or to increase productivity — and they are externally rewarded to deal with the burden associated to achieving that goal. Contrary to professional programmers, end–users program as an orthogonal activity to their primary job function but, nevertheless, both non-technical and technical users receive essentially the same payoff in using the technology in their workplace. On the contrary, there are other contexts in which reward is received qualitatively (such us comfort) instead of quantitatively (money). In this sense, domestic environments are clear examples of the former. Since the payoff is noticeably different, the cost and the risk [13] that the user is willing to assume will be different too. This influences how the user is motivated to overcome the programming learning barriers [74]. To this extent, we conjecture that end-users will invest less effort in programming tasks at home than at work. Following this hypothesis, we firmly believe that domestic environments require new programming paradigms, different from those devised for professional environments.

In addition, the inhabitant's benefit and the environment's goal are ambiguous terms whose meaning may vary from person to person. **Benefit can be a very personal concept (H3)**, as are the burdens users are willing to take for a service variable. Nevertheless, while defining the inhabitant's benefit may be hard even for the inhabitants themselves, almost every one of them will be able to define what they prefer on a smaller scale (e.g. to wake up to music and the smell of coffee). In other words, their benefit may be defined as the sum of all their preferences. Furthermore, what was preferred in the past, may not be so in the future, since people evolve and change habits. Thus, we should consider that there are **multiple and variable preferences per inhabitant (H4)**. In addition, since environments are normally populated by many people, we must consider that **the preferences of multiple inhabitants do not always agree (H5)**.

Social issues

Social groups have their own ways of dealing with conflicts among their preferences. In fact, social groups are characterized (and many times define themselves) by the **communication and sharing among their members (H6)**.

In relation with the individuals, another important psychological issue, especially when talking about programming, is that since the reward is unclear, in

non–compulsory tasks, **anxiety or boredom can lead to abandon** [27] **(H7)**.

3.1.2 Based on the environment's idiosyncrasy

The second characteristic of AmI is that it is based on physical spaces. As stated by Weiser, "computers are vanished into the background" [122], but the background stays and many of their properties will have to be taken into account. The most important for the purpose of this work are related to their physicality, heterogeneity and the fact that they existed before AmI.

Physical spaces

Considering Intelligent Environments as a sort of computing system stresses the differences brought by their physicality. Since the environments are **bounded in their physical extent (E1)**, humans coordinate devices and applications within the same space more often than in different spaces. In addition, since many elements are physical, their number is constrained by the physical space. On the other hand, while their number is smaller, their diversity is greater.

Additionally, physical objects are more significantly bounded to the concept of ownership, which also has strong repercussions on the way humans use them. In this sense, **location and ownership are natural characteristics of elements (E2)** in determining how they are used. Thus, while the number of elements of interest is easily decreased through physical proximity or ownership (e.g. rarely would an application be interested both in a light in one building and a door in another), the number of data types (e.g. lights, doors, messages or calendar events) is common to all of them.

Heterogeneous spaces

The second interesting property of physical spaces is their diversity. There are not only **numerous types of environments (E3)** from a structural point of view, but also **multiple domains of automation (E4)**. That is, there are many different types of environments that can be used for many different purposes.

Inhabited spaces

Finally, the internals of these computing systems **are inhabited (and were inhabited) by people (E5)**.

There are not only people living in them, but they are their homes and cars, and summer houses and work places and parks and thus, as stated by Davidoff et al. [28], **they play a role in group and individual self–definition (e5.1)**. In addition, since they were inhabited before they became a computing system, **there is a *status quo* (e5.2)** already established. This *status quo* can

be appreciated in the way objects/places are used or in the way human beings, due to their social nature, are used to dealing with hierarchical structures in which tasks and responsibilities are spread among their members. Within the house this structure has historically been easy to see: gardeners, housekeepers, butlers and so on [56].

Additionally, human-populated environments are subject to constant changes. These changes can be on different timescales.

Long timescale changes (E6) (space evolution) modify the structure or population of the world, i.e. adding a new light, removing an old one, a new person joining the community or a new intelligent space added to the world.

While the structure or population of the world remains unchanged, some set of preferences may depend on local variations (**short timescale change**) **(E7)**. Thus, even if Xavier exists and there are lighting devices in his house, it does not make sense to apply his preferences when he is not at home.

3.1.3 Based on Ubiquitous Computing's idiosyncrasy

Finally, vanishing computers in the background adds some properties to the environments into which they vanished: the computing capabilities.

Thus, as perceiving and actuating capabilities grow in the environment, through the progress in the development of small low-power hardware, wireless networking, sensor technology and software services, the control possibilities grow exponentially as the combination of possibly perceived contexts with possible actions. That is, there are **many more interaction possibilities (U1)**.

Secondly, the heterogeneity of physical environments is now enriched with the **heterogeneity of hardware and software** embedded in them **(U2)**.

Finally, since not every environment is equipped with the same sensors, actuators or software systems, there can be **huge differences in perception and actuation** capabilities among environments **(U3)**.

3.2 Requirements for end–user programmed indirect–control

Once environments were supplied with perceiving and actuating capabilities, much research effort went into creating "intelligent environments" to actuate on the user's behalf. In this sense, two main trends of research can be identified: *autonomous environments* and *automatic environments*. The former represent those systems that, without user intervention, try to reach the intended goals (such as reducing energy consumption [91] or minimizing the number of tasks the user has to do [23]). Many of these systems are based on "black box technologies", meaning that they are not human–readable, such as neural networks,

Table 3.1: Properties of Ubiquitous Computing environments related to indirect control end–user programming

Based on human factors
Heterogeneity
H1. Different degrees (or absolute lack) of expertise among users.
H2. Heterogeneity of goals and means.
Preferences
H3. Benefit can be a very personal concept.
H4. Multiple and variable preferences per inhabitant.
H5. Multiple inhabitants. Preferences do not always agree.
Social issues
H6. Anxiety or boredom can lead to abandon.
H7. Communication and sharing among inhabitants.
Based on the environment's idiosyncrasy
Physical spaces
E1. Bounded environment.
E2. Location and ownership as natural characteristics of elements.
Heterogeneous spaces
E3. Multiple domains of automation (purpose).
E4. Numerous types of environments (structure).
Inhabited spaces
E5. Already lived in and used spaces.
e5.1. They play a role in group and individual self–definition.
e5.2. There is a *status quo*.
E6. Long timescale change (space evolution).
E7. Short timescale change.
Based on Ubiquitous Computing's idiosyncrasy
U1. Many interaction possibilities.
U2. Heterogeneity of hardware and software.
U3. Huge diversity of possible context and actions.

Hidden Markov models (HMM) or Bayesian networks (BN). Even though HMM and BN may have an interpretable graphical representation, being, thus, interpretable, they are not strictly readable. Automatic systems, on the other hand, represent those other systems trying to replicate the solutions that have been explicitly given by the user. These systems are normally based on "white box technologies" such as rule–based [12] or case–based [15] expert systems, since, as stated by Myers, *the closer the language is to the programmer's original plan, the easier the refinement process will be* [93]. Closer or farther from the end–user's mental plans, those systems pretend to be programmable and, to that extent, they provide a means for creating, structuring and organizing "code". In this

sense, Myers points at rule-based languages as the ones naturally used by users in solving problems [93]. Summarizing, while autonomous environments try to "replace" the user, automatic environments try to extend their control.

While both approaches have advantages and disadvantages, they share the problem of conflicting inputs due to more than one inhabitant, i.e. collisions. While autonomous solutions have a hard time learning from a "noisy" environment, automatic approaches need some clarifying input for what to do in conflict situations.

Our choice has been to leverage the user's control over the environment through an automatic approach, focusing on personal environments and trying to **get as close as possible to the end–user**. In this sense, our work is profoundly influenced by S. Davidoff et al. [28] "social characteristics of home life that should but currently do not influence the development of smart home services", such as that "The house plays a role in family and individual self–definition", "Families are plural. Most systems are singular" or "The *thermostat predicament*: rules don't always agree".

The purpose of the indirect control mechanism is to allow users to "program" behaviors/preferences/implicit interaction into their environments. To accomplish this, we must identify the requirements of the user both as the developer of the programs running in the environment and as the consumer of said programs.

3.2.1 End–user Requirements

R0. Human–centered interaction and flexible needs

Directly related with **H3** and **E5** (see table 3.1), allowing end–users to program their environments is the main requirement of this work.

As stated before, physical places are populated by different people, used to living in them. Thus, any technological improvement of their environment is not only an additional service but a potential change of the *status quo*. For example, conflict resolution is not always a straightforward process; it has to deal with natural hierarchies and particular users' preferences if it is to succeed. A similar problem can be found in defining benefit, especially in personal environments. Contrary to a business process, benefit can be a very personal concept, hard to describe by third parties. As a consequence, we believe that end–users must be able to program their environments in order to define their own preferences and hierarchies, in contrast to most engineer–oriented programming techniques of AmI and other domains.

Table 3.2: Requirements for end–users

Requirements for the End–user	
R0. Human–centered interaction and flexible needs	Directly related with **H3** and **E5**, enabling end–users to program their environments is the main requirement of this work.

3.2.2 Requirements for the End–user as Developer

Allowing the end–users to program the environment requires a programming language in which they can express their preferences, as well as some aids to improve the competence of such non–professional programs.

R1. Expressiveness

As stated by Davidoff et al. as a principle for end–user programming [29] —and directly related with **H3**, **E5** and **E6**— to allow the construction and modification of behaviors must be the first requirement of an end–user development (EUD) system.

Given the high heterogeneity of Intelligent environments (**U1**, **U2** and **U3**), the programming mechanism must preserve a static concept of programming for the user (regardless of what is being programmed or from where). That is, the programming mechanism must be **application–independent (r.1.1)**

In this sense we must analyze how end–users think about programming instead of focusing on how specific applications are programmed. Truong et al. [116] pose that end–users "tended to frame the description of their desired applications in terms of their domestic goals and needs rather than in terms of device behaviors". This conclusion was extracted from a study conducted over a three–week period with 45 participants. The study was composed of two different surveys, each of which described a particular scenario related to capture and access applications, depicted as comic strips. The survey asked participants to explain the applications shown in the scenarios in their own words, and to devise an application of their own.

Promising as the results were, since it only considered scenarios for capturing and accessing audio/video streaming, it is not clear that the same conclusions could be applied to more complex and varied cases. In this sense, Dey et al. [33] conducted a similar experiment but following a broader approach. They found that 56.4% of all the rules developed by the participants involved objects or the state of objects, and that 19.1% of them involved activities. The remaining rules referred to locations (12.8%), time (7.6%) and people different from the subject (4%).

Inspired by Nardi's view, when she stated that *end–user programming languages should be task-specific, language primitives should map to tasks in the domain the user understands* [95], we believe that the system should be designed to deal with all these concepts, with a special effort toward achieving simplicity over the most commonly used (i.e. objects). As noted by Nardi, task-specificity eases users' understanding of what the primitives of the language do, allowing users to develop applications because they can *directly express domain semantics in the high–level operations of the language* [95].

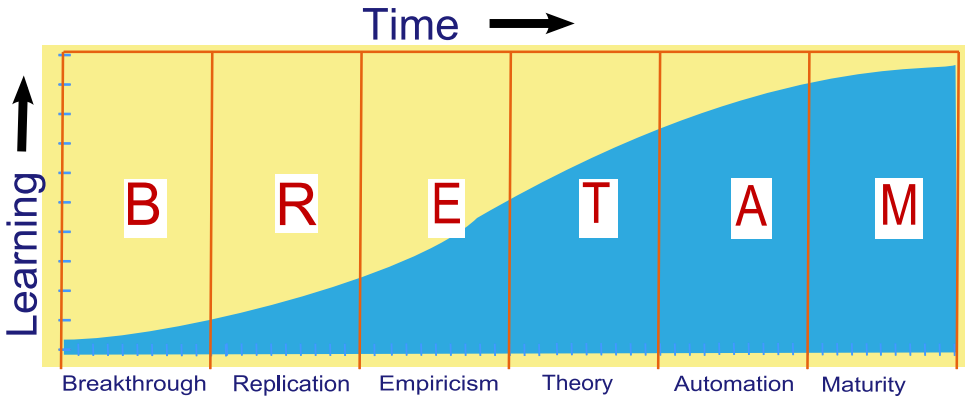
In addition, another important point is how end–users conceptualize the technology behind the house. Truong et al. differentiated three models: System as Effector, System as Assistant and System as Effector-Assistant Hybrid. When they evaluated the CAMP programming tool, they realized that participants were especially partial to the "System as Effector" model for describing scenarios and applications, rather than considering the system as an assistant. This fact was confirmed by the previous Dey et al. study. In this case 98% of all rules concerned present a view of the house as a commanded system. We strongly support this conclusion. In fact, as we will explain, this fact will be a key conditioning point in the design of our expression tools.

In conclusion, we believe that the system must be designed as a commanded system, to match the view end–users have of it.

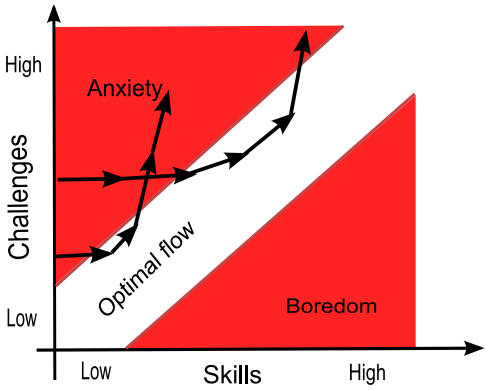
Additionally, given the possible lack of expertise of programmers (**H1**), **simplicity (r.1.2)** is the second requirement. To that extent, stressing how people think about programming and conceptualize the technology behind the house can be considered as having an expression system that is easy and natural.

Natural in the sense stated by Myers et al. [93] as faithfully representing nature or life, meaning that it works the way people expect. When defining programming as "the process of transforming a mental plan in familiar terms into one compatible with the computer" [64] Myers et al. declared that the closer the language is to the programmer's original plan, the easier this refinement process will be [93]. In the same article Myers et al. observe through two studies examining the languages and structures that children and adults naturally use in solving problems, that "an event-based or rule-based structure was often used, where actions were taken in response to events".

Easy, on the other hand, in the sense stated by Greenberg [53] as "removing low-level implementation [so] programmers can rapidly generate and test new ideas, replicate and refine ideas, and create demonstrations for others to try". In the same work he demonstrated, in relation to groupware application development, how a good development tool can break the bottleneck suffered in Gaines' BRETAM —a phenomenological model of how science technology develops over time [42]— (see Figure 3.1(a)). This means that, even when some good ideas or *Breakthroughs* (the B of BRETAM) arise successfully in the field, the constant



(a) Gaines BRETAM software developing model. Taken from [42]



(b) Csikszentmihalyi's notion of flow for analyzing motivational factors in learning. Taken from [106]

Figure 3.1: The two main problems due to the lack of appropriate development tools. 3.1(a) refers to Gaines' BRETAM software developing model [42] in which Greenberg found that a lack of appropriate tools can cause a break in the Replication stage [53]. 3.1(b) refers to the notion of flow, introduced by Csikszentmihalyi [27] to analyze motivational factors in learning and used by Repenning and Iaonnidou as a guide for EUD usability [106]

necessity to deal with hard low–level concepts impedes the *Replication* (the R of BRETAM) and improvement of those previous ideas and hence stops the development process. We believe that some clues for the development of Intelligent Environment applications (professional and end–user development) can also be found here.

Thirdly, to deal with different places, preferences, social structures or concepts (**H2**, **H4**, **E3**, **E4**, **U3**) the end–user requires a **flexible (r.1.3)** means of expression.

This flexibility has to be balanced with the simplicity of expression, paying special attention to short–term effort and long–term restrictions (as in Papert's

ideal of **"low–threshold no ceiling"** [101]) **(r.1.4)**. We believe, like Repenning and Ioannidou [106] —based on the notion of optimal flow of motivation in learning [27] (see Figure 3.1(b))— that "anxiety results if challenges outweigh the skills, while boredom results if skills outweigh the challenges" (**H6**), more especially when talking about personal environments.

R2. Aid for competence

Competence is one of the two requirements stated by Maes for software agents [78]. The lack of expertise of programmers (**H1**) requires some aids to improve the program's performance. Thus, users may want to ask the system for an explanation of its reactions, in order to understand and correct its behaviors, or they will expect the system to understand what they really mean when programming, regardless of small programming mistakes. That is, the system should provide an easy debugging mechanism and programming assistance to minimize the anxiety produced by error situations **(H6)**.

3.2.3 Requirements for the End–user as Consumer

Following the principles of competence and trust stated by P. Maes [78] is of critical importance when analyzing the requirements of the end–user as a consumer. Thus, the system must be trustable and competent. Trustable meaning that the user must feel safe and confident in using the system; competent in the sense that the system performs as expected and adapts to the user's changing needs

R3. Trustable

Trust is especially significant since automated spaces are often personal, thus especially sensitive to mistrust or situations leading to anxiety (**E5**, **H6**). Kay et al. [71] describe how "...when the user wants to know why systems are performing as they are or what the user model believes about them, they should be able to scrutinize the model and the associated personalization processes". In this sense, the system must be **understandable, predictable and traceable (r.3.1)** in order to answer "What has happened", "What is going to happen" and "Why is it happening (or going to happen)", natural questions of non–technical inhabitants (**H1**). Close to Cheverst's et al. concept of *scrutability*, defined as "the ability of a user to interrogate her user model in order to understand the system's behavior" [20].

Secondly, the system must be **respectful (r.3.2)** with the lifestyle choices [29] of the inhabitants and, since the environments play a key role in group and individual self–definition (**E5**), not everything must be automated, just what the user asked to be.

Table 3.3: Requirements for end–users as developers

Requirements for the End–user as Developer	
R1. Expressiveness	Based on Davidoff's principles for end–user programming [29], allows the construction and modification of behaviors as directly related with **H3**, **E5** and **E6**.
r1.1. Application–independence	One of the main problems of current technologies. To keep a static concept of programming (regardless of what is being programmed or from where), derives from **U1**, **U2** and **U3**.
r1.2. Simplicity	Easiness and Naturalness. Get as close as possible to the end-user's programming mental plans to allow non programmers to control their environment **H1**.
r1.3. Flexibility	**H1**, **H2**, **H3**, **H4**, **E3**, **E4** and **U3** require a flexible expression meant to deal with different places, preferences or concepts.
r1.4. "Low threshold, now ceiling"	Papert's ideal [101]. Directly inferred from **H6**, is extracted from [106] and the notion of optimal flow of motivation in learning [27]
R2. Aid for competence	One of the two requirements stated by Maes for software agents [78]. The lack of expertise of programmers (**H1**) requires some aids to improve the program's performance. Easy debugging and programming assistance. Competence is especially necessary due to **H6**.

Finally, Davidoff et al. [29] stated that families' routines suffer breakdowns **(E5)**, forcing a change in preferences or responsibilities within time. Thus, he stated that systems must be **"designed for breakdowns"(r.3.3)**, to deal with them without catastrophic solutions.

R4. Extensibility and scalability

Regarding competence, since preferences are often bounded to physical elements (**E1**, **E2**) that may change along with their owner's lifestyle, a change in the inhabitant's lifestyle (**H4**, **E3**, **E6**) may be reflected in an evolution of the preferences to which the system must adapt. That is, the system must allow for the organic evolution of preferences [29].

R5. Reusability and portability

In addition, preferences are personal choices, bounded to places as long as the person with those preferences uses them. But if the user moves to another space she will probably maintain most of her preferences, trying to adapt them to the structure of the new place (**H4**). In a similar way considering users in their social context (**H7**), they may want to "copy" some of the behaviors observed at a friend's house, and probably the friend will be willing to share them with them, therefore reusability and portability are two requirements of our indirect control mechanism.

R6. Allow simultaneous and changing preferences

Finally, as context changes (**E7**), people join and share environments (**H5**) or preferences grow (**H4**, **E3**), the number of coexisting programs is increased; thus, a flexible manager and coordination mechanism is necessary. In addition, to face the problem of having diverse inhabitants with different preferences, automatic environments need to provide a means through which to create coordination structures, **flexible** enough to adapt to the different social coordination structures of the inhabitants (i.e. hierarchies)

3.3 Implemented solutions and requirements supported

In order to deal with these requirements we have designed an ECA–rule based, multi–agent mechanism, using the Blackboard of P. Haya [62] as an abstraction layer in a multilayer structure that helps us in dealing with some of the requirements. The multilayer structure can be seen in Figure 3.2 and shows how the system described in this thesis is used as the core language and structure of the "end–user implicit interaction" layer. This layer uses the AmIlab Blackboard and Privacy filter [36] as a context layer between it and the world.

We believe that simplifying the language and allowing the decomposition of problems are essential to simplicity in adaptation.

3.3.1 S1. Abstraction layer

As in every intelligent environment, the first step is always that of perception (see Figure 3.2), implemented in our approach through a middleware layer ("the Blackboard") in which every element of the environment, either real or virtual, is represented as an *entity* with *properties* and *relations* between them [60]. Every change in the environment is reflected in the blackboard and vice versa, easing the process of expression (**r1.2**) by removing low–level details of the elements.

Table 3.4: Requirements for end–users as consumers

Requirements of the End–user as a Consumer	
R3. Trustable.	One of the two requirements stated by Maes for software agents [78]. Especially significant since the automated spaces are most often personal, thus especially sensitive to mistrust or situations leading to anxiety (**E5** and **H6**). In order to empower trust some sub–requirements have been defined.
r3.1. Understandable, predictable and traceable.	In personal environments (**E5**) of non–technical inhabitants (**H1**), naturalness is a warrant of trust. Thus, "What has happened", "What is going to happen" and "Why is it happening (or going to happen)" are key questions in need of answers
r3.2. Respectful.	The system must be respectful with lifestyle choices [29]. Not everything must be automated, that is the user's choice (**E5**)
r3.3. Designed for breakdowns.	Extracted from [29]. Since breakdowns occur, both in software systems and in users' routines, the system must be prepared to deal with them without catastrophic solutions (**E5**)
R4. Extensibility and scalability	Allow for the organic evolution of spaces and preferences. Preferences are often times bounded to physical elements (**E1**, **E2**) that change along with their owner's lifestyle. Change of habits, promotions or improvements (**H4**, **E3**, **E6**) are translated into an evolution of the preferences too.
R5. Reusability and portability.	Variable preferences (**H4**) and social interaction (**H7**) require a means to reuse and share the environment's behaviors
R6. Allow simultaneous and changing preferences.	As context changes (**E7**), people co–exist (**H5**) or preferences grow (**H4**, **E3**), the number of coexisting programs is increased, therefore a flexible manager and coordination mechanism is necessary

The blackboard uses a "photographic" representation of the world [60], in which a TV is either ON or OFF and no event (e.g. turning ON) is directly represented but indirectly inferred from the TV changing from ON "in one picture" to OFF in the next one. It also provides a simple API, homogenizing the heterogeneity of elements in a simplified, event–free, function–free, application–independent representation (**r1.1**)

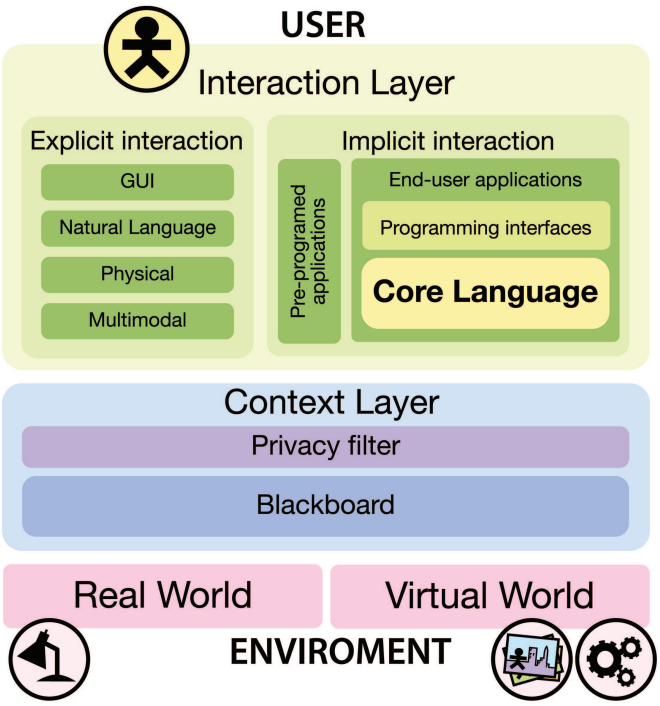


Figure 3.2: AmiLab layers. The World layer (i.e. what is in the environment), the Context layer (i.e. how Ubiquitous Computing accesses it: abstraction [60] and privileges [36]) and the Interaction layer (i.e. how it can be used: Explicitly [2][88] and Implicitly).

3.3.2 S2. ECA–rule language

Over this context layer, in the interaction layer in which all the applications are located, is the indirect control mechanism. This mechanism is designed to allow users to program their environment and it relies on a rule–based core language (see Section 4).

We argue, like many others [15][33][110][113][119][116], that rule–based systems are a feasible approach for codifying user's desires and preferences (**r1.1**). In addition, rule statements can be easily translated to human–readable expressions, therefore they can be understandable, predictable and traceable (**r3.1**) and allow for autonomous learning techniques as well as debugging processes to be applied (**R2**)

Understanding preferences as commands to the environment, we have chosen imperative programming over declarative programming in the form of an ECA–rule based programming language that provides a natural means of expression (**R0**, **R1**) and explanation and supports automatic learning techniques for improvement (**R2**).

Expression

Since the diversity of context and actions can be huge, we should find a balance between the language's descriptive power and its simplicity. This is done by basing our design on two principles: keeping a simple base language and isolating complexity in those special functions that require it.

s2.1. Base–language

Following Papert's ideal of "low–threshold no ceiling" [101], the rule–based language uses a basic language with some expression extensions. The rule–based language is composed of three parts: *triggers* and *conditions*, codifying the context, and some associated *actions*, as an analogy of the natural *"When ... if ... then ..."* structure. Comparators and operators of the conditions and actions, respectively, are kept as simple as possible to create a clean and clear basic programming language (**r1.2**) (See Section 4.1.1).

s2.2. Expression extensions

Some scenarios require more expressiveness, but adapting the whole language to deal with those complex elements brings their complexity to the whole language. Thus, we isolated the complexity to those elements requiring it, so users need only deal with complex concepts in those cases where they are really needed (**r1.3**). Additionally, we tried to express complexity in base language terms, making it easy to understand complex concepts when the basics are known, drawing a reasonable learning curve (**r1.4**).

One of the main problems of expression is time. Not every indirect controlled action is to be executed immediately; some contexts trigger special situations in which the required action depends on some time factor, e.g. "Turn off the oven in 20 minutes" or "if somebody enters the room in the next 5 minutes send me a message". Other research has focused on temporal reasoning by embedding time concepts in the whole representation and reasoning system [6]. This forces the use of events, intervals, date lines or other representation mechanisms at the core of the system, making it especially suited to dealing with time issues, but messy and complicated when dealing with time–independent scenarios.

Of special interest in our approach is how we isolated time constraints by confining them to a special kind of action: the *TIMER* (see Section 4.1.3).

This complex extension is built with the agent's and base–language's structure and increases the expression power considerably, allowing for the definition (besides time–sensitive actions) of composite events and event consumption policies that cannot be defined in traditional ECA–rule based languages (see Section 4.1.4)

Explanation

The second benefit of ECA–rules is that they are understandable and predictable. Also, **tagging changes on the blackboard (s.2.3)** with the agent responsible for the change makes them traceable (**r3.1**) (). This has been used in two ways to create an easy debugging mechanism. First, as a mechanism for scrutability, **allowing users to ask agents about their internal state (s.2.4)**, (e.g. what truth values do its conditions have, which rules are active or how confident is it about the behavior of its actions)

(*scrutability* is defined as "the ability of a user to interrogate her user model in order to understand the system's behavior" [20]);

and second, as a mechanism of **explanation logs (s.2.5)** through which agents show their internal reasoning process with different possible levels of detail (see Section 5.3.1).

Learning

In addition, in order to provide the user with some competence feedback, a **confidence factor** has been added to each action. This confidence factor is increased and decreased through a **reinforcement learning (s2.6)** process and can be used to either activate/deactivate rules automatically or to inform the user of a malfunction (**R2**) (see Section 5.3.2). Furthermore, this variable may be used to characterize the type of error a rule has so, if automatically solved, it will be at the user's command and, consequently, it will not be perceived as an intrusion on the user's control but as a refinement of their expressions (**r3.2**).

3.3.3 S3. Multi–agent structure

While rules are the minimum piece of reasoning information, they are not the minimum reasoning structure. Most tasks and preferences comprise more than one rule, conceptually treated as a single preference. Therefore, we have designed a multi–agent structure to organize rules (see Section 5.1). An agent is the minimum reasoning structure and different agents may be created with different sets of rules. Each agent has three different sets of rules *on–load*, *on–running* and *on–finished* encoded in a file, so it does not matter through which interface the file has been created.

Agents, therefore, are a natural modularization tool (similar to having many butlers that can be commanded) whose potential is increased by representing them in the Blackboard as another part of the context (like a real butler would be), and by combining their strengths with those of the Privacy and Blackboard layers.

s3.1. Modularization

Modularization has been used to tackle many different automatic adaptation problems.

Firstly, by allowing whole packages of rules to be added/removed/modified without affecting the rest of the system (**R4**), it allows the problem of automation to be split into smaller ones, simpler to solve.

When talking about a complex task such as configuring a home to adapt to a wide range of preferences, a modular architecture allows the problem to be handled in a limited, small scale, creating preferences for small domains, situations or elements in a progressive manner. These small and individual tasks are easier to handle than the overwhelming goal of "configuring the home". In this way, the "configuration of the home" is just the result of the combined actions of the "configuration of the parts" represented in the different agents created to deal with the smaller problems.

Besides simplifying the automation task, modularization allows the environment's behaviors to be updated to long timescale changes through the addition or removal of agents without affecting the rest of the system. For example, if an agent is in charge of inferring the location of the inhabitants and a new, more reliable, mechanism is installed, removing the old agent will affect no other part of the system. Since every reasoning unit is independent, the only change will be more accurate location information. Additionally, if a coffee maker is added to the kitchen, an agent could be created to make coffee at 7:00 a.m. from Monday to Friday or to announce when coffee is ready. If the coffee maker is removed, so can the agent.

Secondly, it allows for the spreading of responsibilities among different modules so a failure of a part is not understood as a failure of the whole (**r3.3**), favoring trust trough classification (i.e. a failure of the lighting preferences module will not affect the trust of the security module). This is done similarly to real environments in which a gardener, a broker or a maid will have different domains of action with different associated responsibilities. Emulating this well-known human modularization structure helps the user to find the responsible parts and empowers trust through familiarity.

A modular architecture opens up a natural path to reusability, allowing whole packages to be move or replicated to other spaces (**R5**). Thus, behaviors can be easily exported from one place to another (e.g. "I like the way you control the lights in your living room, can I copy it?"). Grouping behaviors under independent agents, the only remaining task to achieve reusability is to define the isomorphism between the two environments (e.g. "what you call lamp 1 in your living room is called main lamp in mine"). Thus, if users want to export behaviors from one place to another, they only need to copy the agent in charge and establish the correspondence between the elements in the two environments.

This idea is similar to that of the Digital Recipes explained by Newman, Smith and Schilit [97]

s3.2. Blackboard representation

Agents are represented in the Blackboard as another part of the context and, consequently, can be activated and deactivated through it (see Section 5.2.2). This can be done through any direct control mechanism or through the indirect control mechanism presented in this thesis. That is, through other agents. This is specially useful for dealing with short timescale changes in which, even though the structure or population of the world remains unchanged, some set of preferences may depend on local variations. Modular agents can be used as a mechanism for activating whole sets of preferences according to context. Combined with the agent's representation as another element of the world, a modular architecture allows the creation of "meta–agents" in charge of activating/deactivating other agents according to the perceived context, "meta–meta–agents" in charge of changing the changing policies according to context, and so on (**R6**).

s3.3. Multi–layer structure

Finally, agents are represented in the Blackboard as *agent* entities with *status* and *task* properties and the *is_owner*, *located_at* and *affects* relations, linking it with the user that created it, the location it acts in and the elements it affects, respectively. Allowing (but not imposing) the tagging of agents with their owner, location, activity, effects or purpose, allows the activation/deactivation to be done in terms of the natural end–user hierarchies (**R6**) (e.g. when Mom is in the room, their lighting preferences are applied over the child's preferences) (see Section 5.4).

This way of managing preferences is combined with the priority queues of the Blackboard [62] (for dealing with context–independent collisions) and the ownership rights and access policies of the Privacy layer to build a multi–layer filtering structure for users to replicate their natural hierarchies and coordination policies without external restrictions (such as having to relate their preferences to activities).

At this point it is possible to see that most design decisions, from the simplicity of the language to the modular architecture capabilities or the rule-based/event-based structure, are aimed towards the same goal: to make the process of environment automation as easy and natural as possible. These solutions will be explained in more detail in Chapters 4 and 5.

Table 3.5: Implemented solutions and requirements supported

S1. Abstraction layer	Eases the process of expression (**r1.2**) by removing low–level details of the elements Homogenizes the heterogeneity of elements in a simplified, event–free representation (**r1.1**)
S2. ECA–rule language	An ECA–rule programming based language allows a natural means of expression (**R0**, **R1**) It is human readable and thus understandable, predictable and traceable (**r3.1**) Allows autonomous learning techniques as well as debugging processes to be applied (**R2**)
Expression	
s2.1. Base–language	Isolation complexity in special operator keeps base language simple (**r1.2**)
s2.2. Expression extensions	Special operators allow dealing with different and more complex concepts such as time or generality (**r1.3**) The special operators are designed in terms of the base–language, having a reasonable learning curve (**r1.4**)
Explanation	
s2.3. Tagging of actions	Makes the responsibility traceable (**r3.1**)
s2.4. Showing internal state	Makes the reasoning process predictable (**r3.1**)
s2.5. Explanation logs	Makes the reasoning process understandable (**r3.1**)
Learning	
s2.6. Reinforcement learning	To locate malfunctioning rules and possibly suggest improvements (**R2**, **r3.2**)
S3. Multi–agent structure	Rules are grouped in different independent agents
s3.1. Modularization	Spread responsibilities so a failure of a part is not understood as a failure of the whole (**r3.3**) It allows adding/removing/modifying whole packages of rules without affecting the rest of the system (**R4**) It allows moving or replicating whole packages to other spaces (**R5**)
s3.2. Blackboard representation	It allows activating and deactivating agents according to context (**R6**)
s3.3. Multi–layer structure	Tagged with its owner, location, activity, effects or purpose, the activation/deactivation can simulate natural hierarchies (**packagesR6**)

Chapter 4

ECA–rule language

Automated reasoning is probably one of the areas of computing that has received the most attention in the history of computer science. Furthermore, it can be said that it was actually the seed for it, unifying mathematicians, philosophers and linguists in their efforts to define the Truth. Mathematical logic, modus ponens and formal languages are still the basis for most reasoning approaches, e.g. Situation Calculus, Event Calculus, ECA rules, Knowledge Representation and Logic Programming. This diversity of approaches has often been enabled by the necessity of expression of the domains in which we intended to apply automated reasoning. Thus, when new domains are targeted for automated reasoning, new reasoning techniques appear to fulfill their needs.

In recent years, business processing and the semantic web have received special attention, the former pushing improvements in systems' expression capabilities, the latter in the interchange and sharing possibilities of reasoning structures. Argumentation semantics for defeasible logics [51], semantics of composite events [19], interval–based event logics for ECA rules [102] and Rule Markup Languages (e.g. RuleML) are some examples of the improvements derived from the needs of these two domains.

Adaption Hypermedia and web–based systems have exploit the possibilities of rule languages to express intelligent adaptive behaviors in personalized web courseware, pointing out some requirements, close to that of Intelligent Environments, such us reuse, flexibility and ease of use. This is the case of LAG [25], LAG–XLS [111] and GAM [30].

Since every kind of knowledge can be encoded in a language, with its strengths and weaknesses, the first task is finding the language that best describes the needs of our world of preferences, desires and conclusions and allows us to meet our expression, explanation and learning requirements. Expression and learning can be achieved through almost any language, but the explanation requirement

compels us to a language capable of describing its encoded knowledge in a human readable way, thus excluding black-box alternatives such as neural networks.

Allowing users to program the "instinctive reactions" of an environment, in an application–independent manner, is directly related with the inner structure of instinctive reactions as well as with the internal mental processes of human programming. That is, the universal mental patterns used by humans to program. While Minsky elaborates the instinctive reactions as a set of "if–then" rules [83], Myers points at rule-based languages as the ones naturally used in solving problems [93]. Thus, we propose an Event Condition Action (ECA) rule–based language as the core programming language for users to provide the environment with a base–level intelligence. This core language is User Interface (UI) independent —designed to match the internal mental plans of the user— allowing UI–designers to focus just on interaction issues.

ECA–rules, in addition, allow us to encode the most important types of knowledge for context–aware applications. First and most importantly, to express the desires and preferences of the user. Since most of these desires involve the user's will to have something done when some situation arises, the language should be capable of describing the "something to be done" and the "arising situation", in other words, **actions and context**. Secondly, regarding the conclusions, to **describe high-level context from low-level information**.

Besides considering the human natural programming structures, the programming language must take into account the different degrees of skills and challenges, derived from the heterogeneity of users, preferences and scenarios, as well as a completely different concept of duty or payment than that of a professional programmer. In this sense, the absence of an strict duty feeling makes end–users more inclined to abandon under frustrating conditions than employees. In order to avoid abandonment, any alternative designed for end–users must provide an increasing degree of complexity with a very low starting point as in Paperts' ideal of "low–threshold no ceiling" [101] (see Section 3.2.2). Thus, the proposed ECA–rule language is composed of a simple base language, codifying the most basic programming concepts (objects, activities or location [33]) and a set of advanced language extensions (built in the base language terms to ease the learning curve) to deal with more complex concepts, such as generality or time, just when they are needed.

One important issue of context-aware applications relates to when to check the context in order to act as expected. Because supervised environments can grow to considerable size, and given that not every component has the same timing constraints, determining the time intervals for checking the state of the context becomes a challenging problem. Thus we have decided to take an **event-based** approach in which we assume that only a change in the environment can trigger another change in it.

At this point, one of the main questions that naturally arises is: Why not use existing algorithms or production rule systems? Rete [38] is a very efficient algorithm for comparing large collections of patterns and objects, chaining inferences in a fast and low–cost manner. Additionally some rule languages such as CLIPS or JESS already support Rete, one of the most popular algorithms in inference engines, already applied to some Ubiquitous Computing systems such as [125] or [55]. Two problems surround this algorithm when applying it to our research: centralization and certainty. Rete can be used in systems "containing from a few hundred to more than a thousand patterns and objects" [38], but in our system, patterns are distributed along different agents (as we will explain in Chapter 5), each of which must only deal with its own small set of rules. In other words, instead of a huge expert system, ours can be seen as the coexistence of many smaller ones, each of which may be running on a different machine, meaning that the execution time will hardly grow. In addition, languages such as CLIPS or JESS do not allow some of the concepts we want to be expressed (e.g. TIMERS, see Section 4.1.3) and present an unwanted degree of complexity in other concepts we do not use. However, we have borrowed some ideas from Doorenbos [35], replacing the Rete discrimination network with a set of memories and an index, implemented as a hash table. This approach has proven practical when Rete "working memory elements" are fixed-length tuples, and the length of each tuple is short (e.g. 3–tuples), as in our case.

4.1 Knowledge model

Like many database management systems (DBMS), our knowledge model uses event–condition–action rules (ECA–rules) that can be turned into condition–action rules (CA–rules) or event–action rules (EA–rules) as needed. Since the aim of the system is to allow the user to program the environment, reaction rules are preferred to transformation rules [93]. These rules are clearly divided into three parts, triggers (a.k.a. events), conditions and actions (see Appendix A for the grammar), and have been designed to be as concise as possible:

- Triggers: supervised context variables responsible for activating the rule. Only disjunction of primitive events is allowed in the triggers part.

- Conditions: a set of "context variable–value" pairs representing a context state that needs to be satisfied for detonating the action. Only conjunction of conditions is allowed. Disjunction can be codified as separate disjunction–free rules.

- Action: a "context variable–value" pair to be set when, in a triggered action, all its conditions evaluate to true. Several actions are allowed.

These parts are structured according to the following template:

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

While some systems use CA–rules to create reaction rules, the absence of events in the abstract layer forces us to use ECA–rules. As an example of the need for triggers, we can imagine a living room with a TV and a dimmer light with three values: HIGH, LOW and OFF. In this scenario, when turning on the TV, the user wants to set the light to its LOW value if it was on HIGH. This behavior can be encoded into the context TV–ON AND LIGHT–HIGH with the associated action LIGHT–LOW. If no triggers are specified, any change in any variable of the conditions will cause the revaluation of the rule. Thus, if the user sets the light level to HIGH when the TV is ON, all the conditions will evaluate to true and the system will set the light level to LOW, against the user's will. In other words, "If TV = ON and LIGHT = HIGH then LIGHT = LOW" is not powerful enough to express "When TV = ON if LIGHT = HIGH then LIGHT = LOW"

In order to describe our system, we will use as a guide N. Paton and O. Díaz's survey of fundamental characteristics of active databases [104]. Before going into detail on the language, though, let us consider some simple and intuitive examples to introduce our notation.

4.1.1 Let there be light!

The first example is to turn on a lamp when a switch is pushed. In English, the rule would be expressed as *"When the switch is pushed, if the light called lamp_1 is turned off then turn it on"*. In this example, the trigger is the value of the switch, so the rule will be triggered whenever this value changes. The condition is the light being turned on, and the action is to turn it off.

```
switch:interruptor1:value ::
     light:lamp_1:status = OFF
     =>
        light:lamp_1:status := ON
;
```

The condition can also be expressed in negative form:

```
switch:interruptor1:value ::
     light:lamp_1:status != OFF
     =>
        light:lamp_1: status := ON
;
```

Implementing a toggle switch can be expressed using two rules:

```
switch:interruptor1:value ::
     light:lamp_1:status = OFF
     =>
        light:lamp_1:status := ON
;
switch:interruptor1:value ::
     light:lamp_1:status = ON
     =>
        light:lamp_1:status := OFF
;
```

Or, given that the light's status property is binary, it can be expressed in only one rule using the *ASSIGN NOT* operator (see Section 4.1.3).

```
switch:interruptor1:value ::
     =>
        light:lamp_1:status =! light:lamp_1:status;
```

Note that in the previous rule no condition was defined. Thus, every time the switch is pushed, the action will be executed.

It is also possible to combine different conditions. In a second example we will use the same switch to turn on a second lamp if the first one is already turned on: *"When the switch is pushed, if the light called lamp_1 is turned on and the light called lamp_2 is turned off, then turn lamp_2 on"*.

```
switch:interruptor1:value ::
    light:lamp_1:status = ON && light:lamp_2:status = OFF
    =>
        Light:lamp_2:value := ON;
```

Other property types different from binary can be also used. For instance, the following rule assumes that the room temperature can be in a continuous range of integer values. The rule conditions have two parts: 1) the TV is turned on and 2) the room temperature is quite high (more than 25° Celsius). The action is turning on the air conditioning. The resulting rule is *"When turning on the TV, if the temperature is too high, turn on the air conditioning"*.

```
tv:tv1:status ::
    tv:tv1:status = ON && room:tv_room:temperature > 25
    =>
        aircon:tv_room:status := ON
;
```

Additionally, conditions may use not only property values but also relationships between entities, such as *"When Pablo enters the laboratory, greet him through the avatar Maxine"*.

```
person:Pablo:locatedat ::
    person:Pablo:locatedat = room:lab_B403
    =>
        avatar:maxine:say := "Hello Pablo"
;
```

4.1.2 Conditions

The conditions part of the rule codifies the context of the actions, that is, how certain variables of interest of the world should be at the time an event has been detected in order to command the action. For simplicity's sake, we have defined all conditions as a three–part structure of the type $< LHS > < comparator > < RHS >$ (where LHS and RHS stand for left hand side and right hand side, respectively). In addition, only conjunction of conditions is supported. Any disjunction can be codified as separate rules, each them with only a conjunction of conditions. This is done to ease the automatic learning process, so in the underlying layer it is easier to identify which conjunction of conditions is driving to unwanted actions. The basic set of comparators is the following:

- $=$: The *EQUAL* returns true if the LHS value is equal to the RHS value. When comparing a relation with a value, it returns true if any relation of the subset matches the specified value. If comparing two relations, it returns a true value if LHS is a subset of RHS. Even though it may desirable to add additional operators to compare sets of relations, most of them can be built through this one. Thus, checking whether two subsets A and B are strictly equal can be codified in the conditions $A = B$ && $B = A$.
- != : The *NOT EQUAL* returns true when *EQUAL* returns false and vice versa.
- $>$: The *GREATER THAN*, only for integers, returns true if the LHS value is greater than the RHS value.
- $<$: The *SMALLER THAN*, only for integers, returns true if the LHS value is smaller than the RHS value.

- >= : The *GREATER THAN OR EQUAL*, only for integers, returns true if the LHS value is greater than or equal to the RHS value.
- <= : The *SMALLER THAN OR EQUAL*, only for integers, returns true if the LHS value is smaller than or equal the RHS value.

These basic comparators can be easily extended. They are but an example for working with some of the most common contexts we had to deal with. A natural extension of these comparators set should include comparison of strings, sets and algebraic expressions.

All comparators are designed to work with either a property, a relation or a value in their LHS (or RHS). In order to extend the expressive power of the language, a special type of LHS (or RHS) was created to deal with generic conditions (e.g. "Any light of the room"): the *Wildcard*.

Wildcards

The special symbols $*$ and \$ are used to manage and filter sets of entities instead of a single entity in any part of the ECA–rule. They can replace the *entity* or *property* in the pattern *type* : *entity* : *property* described in the introduction (see Section 1.1). Thus, while $light : lamp\_1 : status$ refers to "lamp_1" status, $light : * : status$ refers to every lamp's status or, more accurately, to any lamp status or set of lamps; e.g. the condition $light : * : status = ON$ would be the set of all the lights turned on (evaluating to false if it is the empty set $\emptyset$, and true otherwise). While the event $light : * : status$ can be translated as *if any light changes its status...*, the symbol $*$ acts as a variable holding the set of matching entities. This set can be accessed through the use of \$ followed by the id of the $*$ they have to access. This id is assigned automatically to $*$ from left to right according to the order of appearance in the rule, starting from 0. If a condition has a \$ on the left hand side (LHS), the \$ set would be filtered to just those elements evaluating the condition to true. Thus, the conditions "$light : * : locatedat = room : lab\_b403$ && $light : \$0 : status = 1$" will take all the lights turned on in lab b403 by first obtaining all the lights in b403 and then filtering this set to those turned on. An example of a rule using wildcards can be seen in Rule 1.

4.1.3 Actions

Paton and Díaz describe four options for actions: structure updates (of the database or rule set), behavior invocation, abort transaction or "do–instead" alternative course of action. Since the Blackboard is deliberately function–free, behaviors are implicitly invoked through database changes. In this way, e.g. for sending a message to somebody, no process is invoked, but a message entity is

Rule 1 Example of a rule that shows any message sent to M. Herranz in every available display at M. Herranz's location

```
message:*
 ::
  message:$0:to = person:mherranz
  &&
  display:*:locatedat
       = person:mherranz:locatedat
  &&
  display:$1:status = available
 =>
  message:$0:to -> display:$1
;
```

added to the blackboard (with its properties such as "text" or "sending time") and two relations created from the sender to the message (i.e. "from") and from the message to the receiver (i.e. "to"). Any messenger service will be subscribed to additions of message entities and thus will be indirectly invoked when the new message is added. Whether this service is an external application or another set of rules for another agent (e.g. moving the relation "to" to the living room display since the receiver is currently in it) is not considered in the action. Thus, **behavior invocation is treated as structure updates**. The *do–instead* action, as defined by M. Stonebraker et al. [112], is designed to act as a barrier between the user's command and its actual execution in the database. While this is perfectly true in a DBMS in which the rules are directly embedded in the DB and a manager is in charge of it, our system has no manager and rule engines work as peers in an upper layer. Furthermore, the only way a rule engine in our system has of knowing that some user commanded something is by being notified of the change, and consequently when it is too late to do anything about it. The same situation occurs with the abort transaction. While these kinds of actions have no use in our system, their goals are also valid in our domain. The absence of a system manager does not imply a complete peer relation among actors at every instant: ownership rights and default policies act many times in the manager's place. For dealing with this kind of situation, the Blackboard has two mechanisms to create policies according to these criteria (able to eliminate or modify the users' commands before they are executed) [36][62].

In relation with structure updates, the rule language provides eight basic operations (see Appendix A):

- CE : Creates a new entity in the blackboard representation (such as a new message). The character # is used to generate a unique identifier. It is defined as <*type*> CE <*entity name*> (e.g. message CE msg# will create

a message entity with a random name such as msg150244561412313345)

- AP : Adds a new property to an existing entity. Since the ontology describes two kinds of properties for types, i.e. compulsory and optional, AP is used to add an optional property to an existing entity. If in the same rule, the # character generates the same identifier. Thus, message CE msg# && message:msg# AP image will create a new message and add the image property to that message.

- -> : Adds a relation to an entity (if an inverse relation is defined in the schema, the symmetric relation will be created automatically by the Blackboard). It can either be used as <*entity1:relation*> -> <*entity2*> to create the relation between entities 1 and 2, or as <*entity1:relation*> -> <*entity2:relation*> to create the relation between entity1 and those related with entity2.

- -< : Removes an existing relation. It functions in the same way as ->

- := : Assigns a value to a property. As with ->, it can be used to assign a direct value or the value of another property.

- =! : Assigns the opposite value (only valid for booleans).

- =- : It subtracts from an integer property a direct value or the value of another property.

- += : It adds to an integer property or concatenates to a string property a direct value or the value of another property.

These basic operations can be easily extended. They are given here as an example for working with plain structure context updates.

An example of a simple rule to control access and do some basic high–level context inference can be seen in Rule 2

In addition, each action has an associated *Confidence factor* used to measure (through a reinforcement learning process explained in Section 5.3) the performance of the action. Briefly, if once executed, the action is not contradicted by the user, its confidence factor is increased, and decreased otherwise.

The Timer

Working in the Ambient Intelligence domain, and trying to create a rule–based mechanism to allow users to express their preferences so they can be automated, we had to deal, at some point, with time–dependent actions like “five minutes after I leave do...” or ”if in the next 5 minutes after I leave someone enters do...”. To do this we created the Timer, a special action, executed as a new thread, with

Rule 2 Example of a rule that, when M. Herranz's card is detected in the laboratory door, if M. Herranz was not located in the lab, it opens the door for him to enter, infers that he is now located in the lab and, consequently, increases the number of inhabitants of the lab by one.

```
cardreader:entrance_B403:value ::
 cardreader:entrance_B403:value = person:mherranz:card
 &&
 person:mherranz:locatedat != room:lab_B403
 =>
  person:mherranz:locatedat -> room:lab_B403
  &&
  room:lab_B403:habitants += 1
  &&
  lock:door_B403:status := OPEN
;
```

an ending horizon, an optional set of rules to execute when it is started (with the timer starting as the only implicit event), a set of rules to analyze and execute while the thread is running and an extra set of rules (with the timer ending as the only implicit event) to execute when it is finished (see Table 4.1.3).

Ending time	The time interval (T_w) in which the Timer is active. It can be fixed (e.g. 13:21:15 15th April 2009), relative (e.g. 5 minutes) or infinite (i.e. it will end by other a–priori unknown means)
Concurrence	The number of times that this timer can be running simultaneously
On finished rules	A set of CA rules to be executed when the Timer ends
On running rules	A set of ECA rules active while the Timer is running. They can act on and be based on the Timer status too
On load rules	An optional set of CA rules to be executed when the Timer is started

Table 4.1: Parts of the Timer action

These parts are structured according to the following template:

```
TIMER ending_time concurrence
{
   on_finished_rules
}
{
   on_running_rules
}
{
   on_load_rules
}
```

Since some actions would be executed some time after they were decided to be taken (e.g. I now decide that I will turn off the lights in 10 minutes), a mechanism for turning back the decisions, or changing the time horizon, must be added (e.g. I now decide that I will turn off the lights in 10 minutes, unless somebody enters). Thus the on–running rules may affect not only the world but also the timer thread itself, or use the state of the thread as part of the context through the following actions:

- $TIMER.pause$: The thread continues to be active but the countdown to the *ending time* is stopped.
- $TIMER.play$: The countdown to the *ending time* is started from the point it was.
- $TIMER.stop$: The countdown is forced to end. The on–finished rules are, consequently, executed.
- $TIMER.kill$: The thread is forced to end. The on–finished rules are, consequently, never executed.
- $TIMER.start$: Can only be used as a trigger. It is generated when the timer starts.
- $TIMER.replay$: The countdown is re–started again from 0. The TIMER.start event is generated.
- $TIMER.reset$: The thread is forced to start again. The on–running rules, if present, are executed.

Since timers can be seen as actions taking place in an interval instead of at a particular time (from where the timer was created until it ends), there is a need to specify how many actions can be executed simultaneously, i.e. how many timers should be running at the same time. This is done through the *concurrent*

constraint (no constraint by default). An example of a rule using timers can be seen in rule 3

As we explain later, this timer structure can be used to express complex events using atomic ECA rules (primitive event conjunction is only allowed) with timers. Additionally, events (as well as conditions and actions) can be enriched with the use of *Wildcards*.

Rule 3 Example of a rule turning on the alarm if the main door is opened for 5 minutes. Note that since no on–load rules are needed the optional last part, corresponding to the on–load rules is omitted.

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

4.1.4 Events

While Paton and Díaz identify events in DBMS as *structure operations*, *behavior invocations*, *transactions*, *user–defined* or *exceptions* [104], we just consider (for the same reasons listed in the actions section) the first, being indistinguishable in our system from behavior invocations, user–defined events or exceptions. Transactions, on the other hand, as the blackboard is implemented right now, are not identifiable as events.

Event type

ECA rule systems distinguish between raw/atomic/primitive and complex/composite events, defining the former as “an instantaneous, significant, atomic occurrence”, and the latter as “built from occurred atomic or other complex event instances according to the operators of an event algebra” [103]. While complex events are strictly necessary for expression, we can view them as the result of a reasoning process that can be encapsulated into an ECA rules chain in which all the events

present in the rules are raw. We will call these kinds of rules, having only a disjunction of atomic events, atomic ECA rules. In this way, an atomic ECA rule may trigger other ECA rules and so on, ending up in actions over the Blackboard. The result of this chain is a complex combination of atomic events. Since ECA rules have not only events, but also conditions, **the process of creating complex events can be enriched with extra constraints derived from the ongoing context** in a way that a traditional event algebra cannot. In other words, composite events can be codified in a chain of ECA rules launching ECA rules as a reasoning process rather than as an event algebra. The *initiator* and *terminator* events described by Paschke [102] in his Interval–based Event Calculus can be found in the ECA rules' chain in the events triggering the first and last rules of the chain, respectively.

To carry out this iterative process we use the timer action described above. Actually, timers can be seen as de facto rule engines. They already have a set of on–running rules and on–finished rules. If needed, a set of on–load rules can be added or codified as on–running rules with the $TIMER.start$ as trigger if we want them to be executed when the action $TIMER.replay$ is executed. We will describe now how event algebra, interval definitions and consumption policies are obtained and improved with this mechanism.

Event algebra

Event operators vary between systems, but the most common are discussed in a number of articles [102][104][19]. These operators are resolved through timers in the following way:

- **Sequence** : $(E_1; E_2)$ occurs whenever E_1 occurs before E_2. Following our semantic proposal, this would be codified in an ECA rule with E_1 as the only event, no condition and a timer as action. The timer will have no on–finished rules and only one on–running ECA rule with E_2 as the only event. T_w denotes the maximum time window in which $E1$ and $E2$ are understood to compose $(E_1; E_2)$. $C_{(E_1;E_2)}$ and $A_{(E_1;E_2)}$ are the set of conditions and actions, respectively, of the original ECA rule with $(E_1; E_2)$ as an event:

  $E_1 :: => TIMER\ T_w\ \{\ \}\ \{E2 :: C_{(E_1;E_2)} => A_{(E_1;E_2)}\}$

  The consumption policies (whether E_2 should be detected only the first time or every time it occurs in T_w) will be explained in Section 4.1.4

- **Disjunction** : $E_1 or E_2$ occurs when either E_1 or E_2 is detected. This composite event can be codified in our system in the following way:

  $E_1\ ||\ E2 :: C_{E_1 or E_2} => A_{E_1 or E_2}$

- **Conjunction** : $E_1 and E_2$ occurs when both E_1 and E_2 are detected in T_w. This may be codified in the two rules codifying $(E_1; E_2)$ and $(E_2; E_1)$:

  $E_1 :: => TIMER\ T_w\ \{\ \}\ \{E2 :: C_{(E_1 and E_2)} => A_{(E_1 and E_2)}\}$
  $E_2 :: => TIMER\ T_w\ \{\ \}\ \{E1 :: C_{(E_1 and E_2)} => A_{(E_1 and E_2)}\}$

- **Closure** : $closure(E, T_w)$ is signaled only the first time E occurs in the T_w interval. Note that the concurrence parameter of the timer is set to 1:

  $E :: => TIMER\ T_w\ 1\ \{\ \}\ \{\ TIMER.start :: C_{closure(E,Int)} => A_{closure(E,Int)}$ $\}$

- **History/Periodic** : $times(n, E)$ is signaled when E occurs n times in T_w:

  $E :: => TIMER\ T_w\ 1\ \{\ \}\ \{\ E :: => TIMER\ T_w\ 1\ \{\ \}\ \{\ ...\ E\ C_{times(n,E)}$ $:: => A_{times(n,E)}\ ...\}\}$

  The concurrence parameter of the timer can be set to 1 if we want $times(n, E)$ to occur only when the n^{th} event E occurs. If we do not set any concurrence constraint to the first timer of the chain (leaving the rest of the timers with concurrence 1) $times(n, E)$ will occur every m^{th} time E occurs with $m \geq n$. More variants can be obtained by varying the concurrence parameters of the timers or/and using the TIMER.kill action in the on–running rules.

  Notice that conceptual recursion (i.e. timers launching timers) is possible as long as we have an upper bound (n in this case), since self–referencing is not allowed in this semantic yet: i.e. a timer can launch another timer, but cannot launch itself. The implications of adding self–referencing in terms of simplicity and flexibility must be further studied.

- **Aperiodic** : $Ap(E_2, E_1, E_3)$ is detected when E_2 occurs within the interval defined by E_1 and E_3. In this case a timer is initiated with the raw event E_1. If E_2 is then detected, a new timer is launched that, on any E_3 detection, will detect $Ap(E_2, E_1, E_3)$

  $E_1\ ::\ =>\ TIMER\ T_w\ \{\ \}\ \{\ E2\ ::\ =>\ TIMER\ T_w\ \{\ C_{Ap(E_2,E_1,E_3)}\ =>$ $A_{Ap(E_2,E_1,E_3)}\ \}\ \{\ E_3\ ::\ =>\ TIMER.stop\ \}\ \}$

- **Not** : $not(E)$ detects the non occurrence of E during the T_w interval. This is codified by a timer, killed if the event E occurs. The on–ending CA rules are consequently executed after T_w only if E has not occurred. The execution of the timer depends on the conditions initiating the interval. Thus we will only describe the timer.

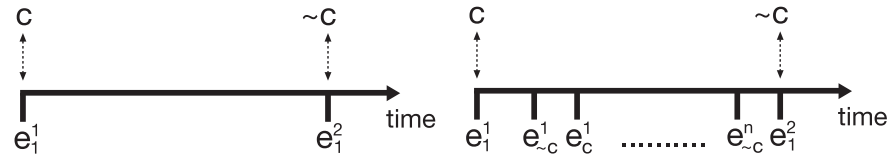
  $TIMER\ T_w\ 1\ \{C_{not(E)} => A_{not(E)}\}\ \{\ E :: => TIMER.kill\}$

In most event algebras, every operator combines raw events or composite events in a particular manner discussing whether or not the compositions detect what they are supposed to detect or they are powerful enough. As an example, Galton and Augusto [47] pointed out at the inadequacy of Snoop [19] detection–based composition to differentiate between composite events E_1;(E_2;E_3) and E_2;(E_1;E_3), driving Snoop to evolve to an interval based semantics based on occurrences instead [18]. But all of them share a characteristic: they use a context–independent composition of events, meaning that they compose events considering just other events. As an example, when Snoop [19] analyzes how the composition of events is affected by context, it describes context in terms of events (e.g. whether an event occurrence is repeated or preceded by some other event occurrence). Thus, an event $E_1; E_1$ in which the initiator E_1 must occur when the condition C is true (i.e. c) while the terminator E_1 must occur when C is false (i.e. $\sim c$), cannot be defined just by event composition and has to use the conditions of the ECA–rule (in those systems in which this is possible) to create time dependent conditions.

Lets explain this example with some more detail. If at the time the initiator event occurs (T_i) c must be a fact, and at the time the terminator event occurs (T_t) $\sim c$ must be a fact, it is possible to infer that an event $E_{\sim c}$ has occurred at a time T_n, $T_i < T_n < T_t$. Thus, in a context–independent event composition, we can try to codify the composite event as $E_1; E_{\sim c}; E_1$. But if an event E_c occurs at time T_m, $T_n < T_m < T_t$, the composite event will be incorrect (since the terminator will be detected in a context in which $\sim c$ is not a fact), so the composite event has to be codified as $E_1; E_{\sim c}; E_c; E_{\sim c}; E_1$. The process continues as long as we continue to accept the possibility of E_c and $E_{\sim c}$ happening between T_i and T_t, giving birth to infinite rules. While this is a simple case in which initiator and terminator depend on a common condition, it is good enough to illustrate the problem of context–dependent events (see Figure 4.1).

Temporal reasoning allows to codify these conditions as conditions of the ECA rule (rather than as conditions strictly associated to a particular events) referring to the time in which the events T_i and T_t occurred. On the other hand, the conditions for the events are merged with the rest of the conditions of the rule, breaking the separation of these two conceptually different sets of conditions. This merging may affect to the easiness of working with complex context–dependent events scenarios in which, for example, different consumption policies are established for each event of a composite event. Thus, in order to apply a policies to a particular event, they have to be applied, one by one, to each of its conditions, disseminated through the ECA–rule's condition set.

The events of the example can be codified with timers in the following rule, where $C_{E_1;E_2}$ and $A_{E_1;E_2}$ are, respectively, the conditions and actions associated with the composite event:



(a) Context–dependent event composition

(b) Context–independent event composition

Figure 4.1: Comparison between context–dependent and context–independent event composition. Since in the latter only events (and no conditions) can participate in the construction of composite events, in order to detect the event of *A)*, an arbitrary long succession of $e_{\ c}$ and e_c must be defined in *B)* as part of the composite event

$$E_1 :: c => TIMER\ T_w\ \{\}\ \{\ E_2 :: \sim c\ \&\&\ C_{E_1;E_2} => A_{E_1;E_2}\}$$

Interval definition

The natural structure for timers is to define *relative* intervals, i.e. one range extreme is defined by an event (the initiator) while the other is a time value relative to the initiator. Since the maximum T_w of the timer can be set to $infinity$, infinite intervals can also be defined or ended according to context instead of time values, since the action $TIMER.stop$ ends the timer, as in the $Ap(E_2, E_1, E_3)$ operator described in Section 4.1.4. Finally, regarding an interval in which both extremes correspond to time values, there has been some discussion about whether the initiator should be treated as another event (subscribing to the "event" *5th January 2009 11:23:00*) or as an internal variable, in which each agent has its own internal calendar and clock. While the former option pleads for a complete representation of the world in the Blackboard (included time), the latter reduces the communication load with the Blackboard.

We have discussed the three *absolute* intervals (i.e. *numeric values*, *events* and *infinite*) and one *relative* traditionally considered in the literature. In addition to these types of intervals, some AmI domains need not just to define an interval, but to **dynamically change the interval according to context**. As an example, while developing a demo (combining this reasoning system with D. Molyneaux's steerable projection system [84] in an AmI environment) in which the home guided the user in cooking by displaying messages on objects as they were needed (e.g. when the water is boiling a "danger very hot water" message was projected on the pot), we found a problem with boiling an egg. The egg must be in boiling water for 12 minutes, thus, the initiator would be the conjunction of "water boiling" and "egg in the pot" $W_b and E_p$ and, naively, the terminator would be 12 minutes relative to $W_b and E_p$. The problem is that these 12 minutes are not only relative to the initiator, but also to the fact that the egg remains in

the pan and the water continues to boil. If any of these facts are contradicted, the interval is no longer valid. Conversely, it is no longer invalid either: the egg has spent some time in the boiling water and the composite event "egg boiled" needs now less time to occur. In other words, the interval should be defined as *12 minutes after the egg is placed in the pot and the water has begun to boil*, but not any 12 minutes, 12 minutes of boiling water with the egg inside the pot; otherwise, time "does not count". This example can be expressed as *"When I put the egg in the boiling water, count 12 minutes, then turn off the stove. If during the 12 minutes I remove the egg or turn off the stove, stop counting. If both the egg is in the water and the water is boiling, continue counting"*. This example is codified in rule 4 with a TIMER as its action.

This kind of *dynamic* interval (either *relative* or *absolute*) can be expressed with our timers (formerly created to deal with them). This is done through the actions defined in Section 4.1.3. Further study is required regarding the need for arithmetic modifiers of the $TIMER$ time (e.g. $TIMER.time += 1m$)

Consumption policies

One of the most common problems in detecting composite events is the consumption policy of raw events. This term is typical of systems in which events are represented in the database and, once used, have to be dispatched somehow. Some authors [19] refer to it as *context* but, even though we do not model events in the Blackboard, we prefer to use the term *consumption policy* instead of the overused *context*.

When detecting a composite event, repeated occurrences of the same raw event may lead to a situation in which the composite one can be detected in various ways. Thus, $(E_1; E_2)$ in the sequence E_1, E'_1, E_2, E'_2 can be detected in four possible ways. Furthermore, we can consider all possible combinations of these four ways as valid detections for some applications.

Traditionally, four different policies have been described. We will analyze them, explain how to model them through timers, and propose a way to extend timers to increase their expressiveness

- **Recent**. Which considers only the most recent occurrences of raw events. Thus $(E_1; E_2)$ in our previous example would be matched by $E'_1 E'_2$. It is important to note that, since our system processes events as they occur, the first terminator to occur would always be the most recent (the only one, in fact). So several occurrences of terminators are only possible when the composite event triggers a rule which takes time to execute, e.g. "5 minutes after $(E_1; E_2)$ do A".

  Recent policies are described with a timer with a not constrained *concurrence* factor, so a timer is launched for every event occurrence E_1. Adding

Rule 4 Example of a rule codifying *When the egg has been boiling for 12 minutes, turn off the stove*

```
device:pot:contains
||
device:pot:boiling
::
 device:pot:contains = egg:egg
 &&
 device:pot:boiling = 1
 =>
  TIMER 12m 1
  {
    device:stove:status := 0 ;
  }
  {
    device:pot:contains ::
      device:pot:contains != egg:egg
      =>
       TIMER.pause
    ;
    device:pot:boiling ::
      device:pot:boiling != 1
      =>
       TIMER.pause
    ;
    device:pot:contains
    ||
    device:pot:boiling
    ::
     device:pot:contains = egg:egg
     &&
     device:pot:boiling = 1
     &&
     TIMER.pause
     =>
      TIMER.start
    ;
  }
;
```

an on–running rule to the timer of $not(E_1)$ (see Section 4.1.4) guarantees that only the most recently launched timer is actually alive. Thus, any time E_1 occurs, a new timer is launched and previously launched timers are killed. This is codified as:

$E_1 :: => TIMER\ T_w\ \{\}\ \{\ E_1 :: => TIMER.kill\}$

Finally, in order to implement the recent policy, it is necessary to include in the previous rule the detection of the last occurrences of E_2.

$E_1 :: => TIMER\ T_w\ \{\}\ \{\ E_1 :: => TIMER.kill\ ;\ E_2 :: => TIMER\ T_w\ \{C_{(E_1;E_2)} => A_{(E_1;E_2)}\ \}\ \{\ E_2 :: => TIMER.kill\ \}$

- **Continuous**. Which considers a sliding window and starts a new composite event with each primitive event that occurs. It is the most natural policy for timers, starting a new composition with every initiator and ending them with the first occurrence of a terminator. In our example, two composite events may be signaled, E_1, E_2 and E_1', E_2. For this policy, timers do not have a *concurrence* constraint and, after detection of the terminator, they end the timer $TIMER.stop$ (in addition to the actions associated with the composite event).

  $E_1 :: => TIMER\ T_w\ \{C_{(E_1;E_2)} => A_{(E_1;E_2)}\ \}\ \{\ E_2 :: => TIMER.stop\}$

- **Chronicle**. Which consumes raw events in chronological order. Having no method to self–reference timers, this policy cannot be currently expressed. Since events are processed as they appear, the only way to express E_1, E_2 and E_1', E_2' in our example, is to create a timer on E_1 occurrences that, in the on–running rules, launches itself again on E_1 occurrences (in addition to the actions associated with E_2). Additionally, some sort of communication between launched timers is needed. As an example, the first occurrence of E_1 will launch a timer T, waiting for E_2 to occur. If before E_2 occurs, a second instance of E_1 is signaled, a replica of T must be launched. This replica T' is not to process any E_2 event until its parent T is done. Thus, a communication mechanism between related timers is needed to define this policy.

- **Cumulative** Which is detected once for all occurrences beginning in the first initiator and ending in the last terminator: E_1, E_2' in our example would be the only composite event detected. The advantage of being able to define different policies for each event is that we can define intermediate policies, in which some events follow different policies than others. Cumulative can be expressed defining a *continuous* policy for E_1 with a concurrent factor of 1 and a *recent* policy for E_2.

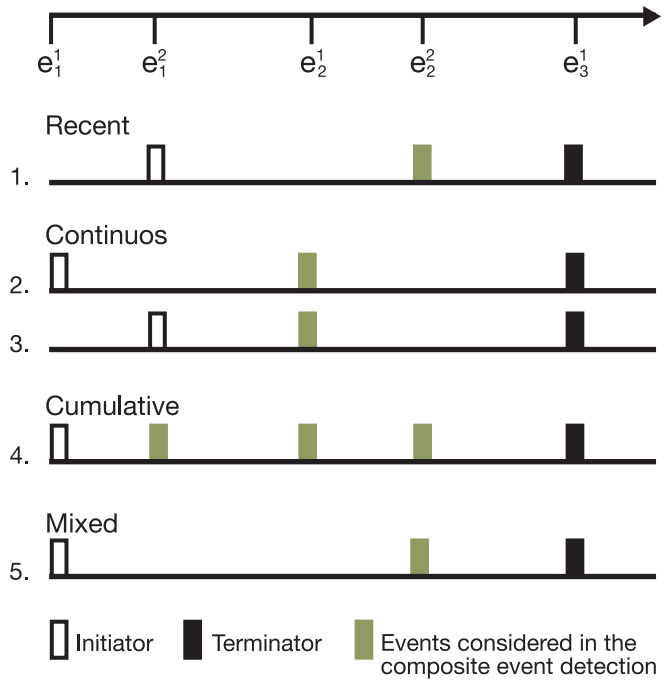


Figure 4.2: Illustration of consumption policies for composite event detection. Various consumption policies are compared with a "mixed" policy in which the composite event is designed to use the first instances of the initiator and terminator event but the last instance of the in between events. Illustration inspired by [19]

$E_1 :: => TIMER\ T_w\ 1\ \{\}\ \{\ E_2 :: => TIMER\ TIMER.time\ \{\ C_{(E_1;E_2)}$ $=> A_{(E_1;E_2)}\ ;\ \}\ \{\ E_2 :: => TIMER.kill\}\ \}$

In this way, a timer is started only with the first initiator, while the terminator is described with the *recent* policy structure described above. Note that the window time of the latter is set to the remaining time of the former.

The advantage of our mechanism is that policies are not globally established, but assigned to each raw event. For example, in the sequence $(E_1; E_2; E_3)$, it can be stated that E_1 and E_3 use the *recent* policy while E_2 uses a different one.

$E_1 :: => TIMER\ T_w\ 1\ \{\}\ \{\ E_2 :: => TIMER\ TIMER.time\ \{\ C_{(E_1;E_2;E_3)}$ $=> A_{(E_1;E_2;E_3)}\ ;\ \}\ \{\ E_2 :: => TIMER.kill\ ;\ E_3 :: => TIMER.stop\}\ \}$

Figure 4.2 illustrates the different consumption policy previously explained, including a mixed one corresponding to the above rule.

The communication between the timer's launchers and launched timers is still an open issue and, while we are sure of the need of a communication mechanism (e.g. specifying if a timer A must continue when the timer B that created it is

no longer alive or specifying a new action $TIMER.parent.kill$, *pause*, etc.), the consequences and possibilities must be further studied.

4.2 User expression

The diversity of user goals, environments, expertise and preferences requires a flexible expression mechanism, with a simple base–language and an increasing complexity of Papert's "low–threshold no ceiling" ideal [101] (see Chapter 3). In this sense, the ECA–language exposed above presents a simple and natural starting point, following the most fundamental concepts of natural programming in an easy to understand and use programming language. This first programming step is designed to allow users with little or no programming knowledge to gain some control over their environment.

All the complexity required to deal with more subtle or intricate concepts is crammed into specific operators in such a way that, while the base–language remains complexity–free, users can increase their control possibilities along with their skills. To make the growing process easier, these operators are built following the same design principles and programming concepts of the base–language (see figure 4.3).

In addition to the expressiveness they provide by themselves (i.e. wildcards allow general rules to be expressed while Timers provide a tool for creating time dependent actions), they have been used to create templates to translate complex concepts into the language without having to add any further complexity to it, improving in some cases the expression capabilities of traditional ECA–rules based languages. This is the case of Timers in relation to event composition, in which the templates presented for event algebra and interval definition allow UI designers to directly translate end–user expressions into the ECA–rule language.

In order to measure the naturalness of the ECA–rule language for end–user programming, we have conducted and end–user study

A user study has been conducted to measure the adequateness of the triggers–conditions–actions structure and the event–free representation of the world for end–user programming.

4.2.1 End–user study

The study was conducted over 30 Spanish speaker subjects, each of which was e-mailed with a short description of an hypothetical smart home (described through some plans)*. In order to measure the naturalness of the language we categorized participants into two groups to compare their results: those with programming experience (P) and those without it (NP).

*Questionnaire, plans and videos can be accessed from http://amilab.ii.uam.es:8080/encuesta

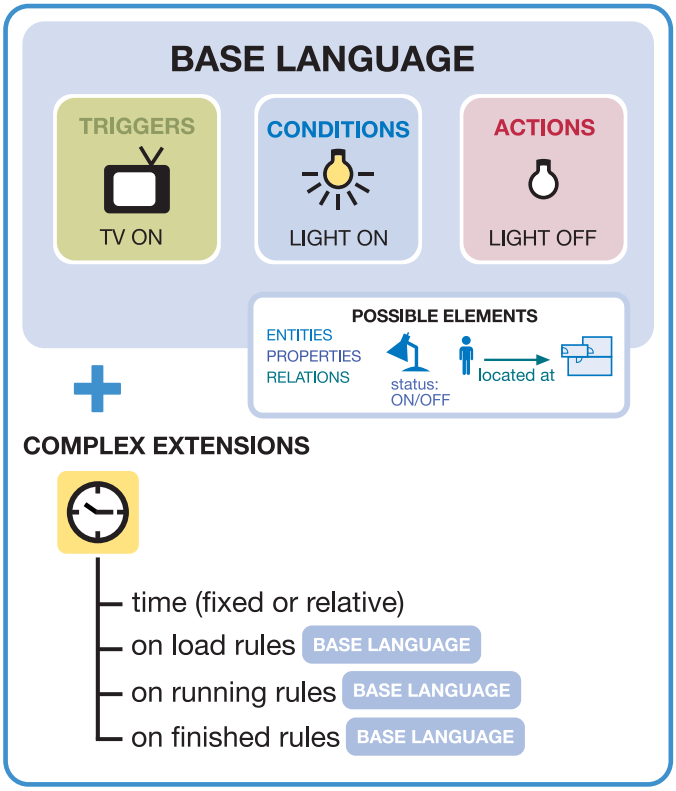


Figure 4.3: Triggers–Conditions–Actions structure of the base language and description of the *TIMER* complex extension.

Participants were introduced into Intelligent environments by being told that the home understands rules constructed by filling a template with three boxes: one for triggers (WHEN), one for conditions (IF) and one for actions (THEN), as long as they refer to the elements present in the plans. Triggers where briefly described as "punctual changes in the environment causing our desire for something to be done". Conditions, on the oder hand, were described as "the set of states in which certain elements have to be for us to desire something to be done". Finally, the actions were described as "the changes we desire to be done". In addition they were provided with two simple examples as part of their training: *"When the TV is turned on, if the sofa is occupied, the light level of the room is high and the light is on, then turn off the light"* and *"When a window is opened, if nobody is at home, then show in my mobile phone screen the image of the living room camera"*. Common sense and lack of fears were finally recommended.

Once naively introduced to the system, participants were asked to perform a series of exercises. The main part of the exercises consisted in writing the appropriate rules to codify the automatic behaviors shown in 5 animations. The animations show scenes recorded in the house in which the environment was

automatically acting according to some context (e.g. turning on and off the lights as people enter or leave the rooms or notifying the user if, when entering the house, the keys were not left in the key–holder). Animations were accompanied with a short description of the type "This animation shows a recording of the kitchen in which the house has been programmed to prevent fires from remaining on when no object is using them" (See figure 4.4). While any kind of verbal description may bias the users, we found it necessary to focus them in the same task since the same video, formerly without description, resulted in different users describing different scenarios of different complexity. While this showed us that different degrees of expertise and background result in different ways of identifying and solving problems it was unacceptable to measure the language naturalness for users tackling the same problems.

Thus, in summary, participants were asked to use a semi-formal grammar splitting each rule in three parts. How to fill each part was leaved to their creativity but constrained to the name and properties appearing in the plan. Therefore, a valid sentence could have been "WHEN the NUMBER OF OCCUPANTS is zero IF KITCHEN_LIGHT is ON, THEN turn KITCHEN_LIGHT off". Uppercase words represents grammar compulsory tokens, and the lowercase ones are free choice tokens. It is worth noting that this semi-formal grammar has been devised in such a way that produced sentences that can be directly translated to our rule base language.

Answers were evaluated by an expert to measure: (I1) how well users stick to the grammar, using only elements present in the plans, (I2) how well they identify triggers and conditions, separating them in different sets and (I3) how well did they assign these sets to their respective boxes. Their performance was measured in a three value scale (3 for good, 2 for medium and 1 for bad). Thus, I1 indicates the naturalness of context representation, that is, how hard is for the end–user to codify her rules being able to use just the properties and relations of a–priori fixed elements with fixed names. I2 and I3, on the other hand, measure the adequateness of the triggers–conditions–actions structure. I2 by measuring how well end–users differentiate between elements acting as triggers and those acting as conditions in the rule, meaning that conditions and triggers were correctly separated by the user into two different sets. In other words, I2 measure how natural is to end–users the conceptual separation between triggers and conditions. I3, on the other hand, measure how good was the association of these sets to the triggers and conditions boxes provided, that is, how natural are the "trigger"/"when" and "condition"/"if" words to categorize those sets.

A number of results were extracted from this part of the study. Firstly, even though Programmers (P) performed better than non–programmers (NP) in I1, there is no significant statistical difference between both groups, obtaining a p–value of Mann–Whitney U test of 0.17 due to 64,62% of P obtaining a 3

Figure 4.4: Screenshoot of an exercise of the end–user study. It shows the animation and plans of the elements involved in the animation in the middle of the web page and a short description of what is the video showing below it.

in their evaluation, against a 26,15% with a 2 and 53.75% of NP obtaining a 3 against 33,75% obtaining a 2. Secondly, identifying and separating triggers from conditions (I2) resulted an easy task for both groups: 87,50% of NP obtained a 3 (2,81 $\pm$ 0,53 in average) as 90,77% of P (2,88 $\pm$ 0,41), showing no significant statistical difference between their performance (p–value of 0.68). Finally I3 has shown to be more complicated for NP users: 57,50% obtained a 3 against the 38,75% with a 1 (2,19 $\pm$ 0,96), being statistically significant (p–value of 0.0029) that P users performed better in it (80,00% of P obtained a 3).

The great majority of tasks were solved using just one or two rules (both by NP and P), while only one participant used more than three rules to encode a task, while every scenario was encoded using less than three minutes on average (see Figure 4.6).

In addition, users were asked to create two more scenarios of their own will, defining the elements they need for them and creating their corresponding rules. Since I1 resulted much better in this case than in the scenarios we provided. we

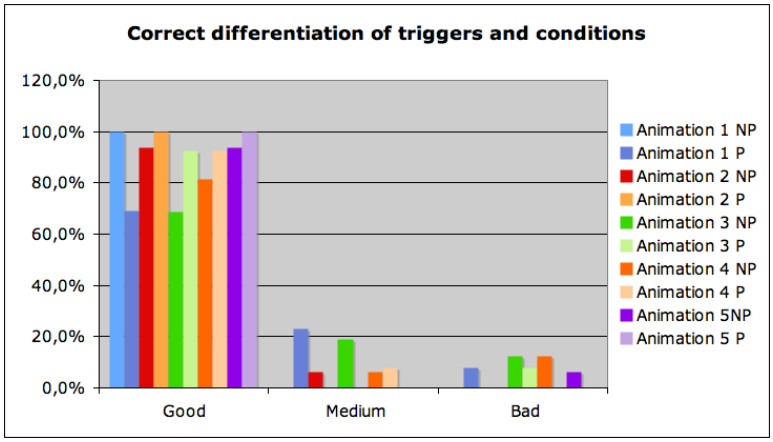
Table 4.2: I1 (Grammar), I2 (Differentiation of Triggers/Conditions) and I3 (Separation of Triggers/Conditions) results (in % for G=Good, M=Medium, B=Bad) for end–users with (P) and without (NP) programming knowledge and their corresponding Mann–Whitney U test p–value.

	I1			**I2**			**I3**		
	G	M	B	G	M	B	G	M	B
NP	63.75	28.75	7.50	87.50	6.25	6.25	57.5	3.75	38.75
P	60.00	27.69	12.31	90.77	6.15	3.08	80.0	9.23	10.77
p		0.17			0.68			0.0029	

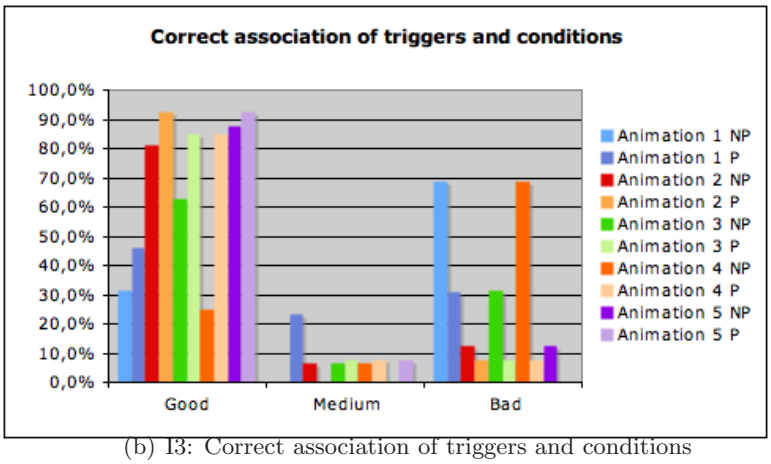
conclude that the bad performance of I1 was due to the unfamiliar home we were presenting and the inability of users to familiarize with a complete house within minutes. I2 and I3 present more interesting conclusions (See Figure 4.5). First, that **the separation between triggers and conditions can be made naturally both by programmers an non programmers**, allowing as to define such separation as natural for programming despite the programming skills of the programmer. Second, that **the Spanish words for "when" and "if" are semantically close (as they are in English) and may lead to misprints among users unfamiliar with the inflexibility of computing languages** e.g. a sentence such as "when I enter the house, if it is empty..." can be also expressed in natural language as "If I enter the house when it is empty...". Thus, **even though triggers and conditions are easily differentiated we should find a more natural way to classify them**.

□

While this language provides the necessary means of expression, an end–user programming system requires more than that. In this sense, complex preferences may be composed by more than one rule, users may have preferences over different domains of control, or many users, with different preferences, may share a single environment. Therefore, besides a language for expression, they will need a mechanism for organizing, structuring and managing the rules they have created. This mechanism will be analyzed in the following chapter.

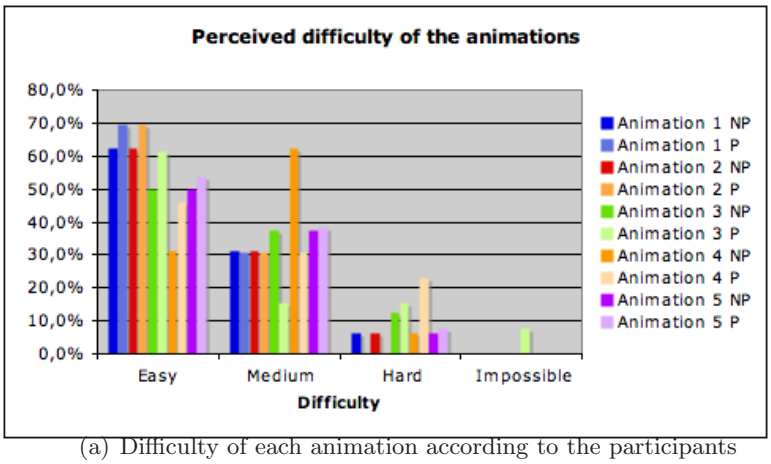


(a) I2: Correct differentiation between triggers and conditions

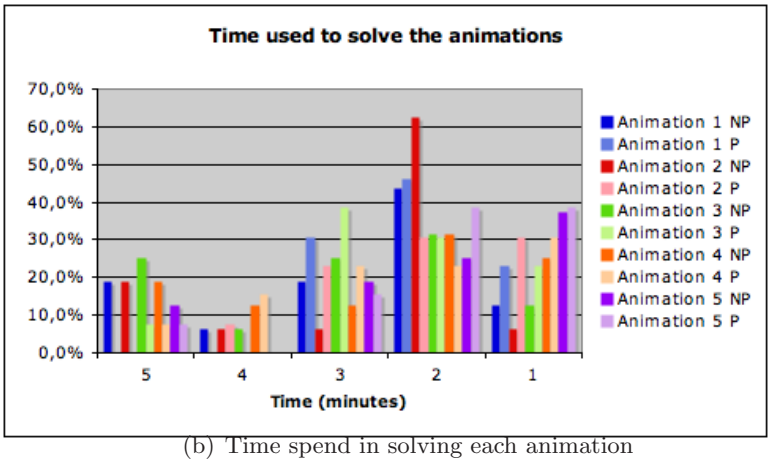


(b) I3: Correct association of triggers and conditions

Figure 4.5: Graphs showing the results of our study for trigger–condition differentiation and association for end–users with and without programming background (P and NP respectively).



(a) Difficulty of each animation according to the participants



(b) Time spend in solving each animation

Figure 4.6: Graphs showing the perceived difficulty and time spent in solving the animations both by end–users with and without programming background (P and NP respectively).

Chapter 5

Multi–agent Structure

5.1 Distributed reasoning

Given that the grammar only defines the comparators, and actuators —leaving the definition of entities to the blackboard— the addition of new types of elements to the environment does not pose any problems to the language, with new elements being integrated smoothly in the rule-based system. This was proven in the Phidgets, anthropomorphic figures and Steerable projection demonstrators Sec 6.3.2, 6.3.1 and Sec 6.3.3 respectively, as we will show in Chapter 6.

On the other hand, due to the increase in controllable elements or to the number of user preferences, the number of rules present in the environment may grow considerably, resulting in a set of preferences that is difficult to manage. In addition, even though the rule is the basic programming unit, tasks or preferences will be, most of the time, composed of many rules. This is represented in our system by *agents*.

An agent is a generic software structure holding and executing a set of rules, irrespective of the UI or UIs used to program them. They are a way for users to organize and manage preferences, rather than Artificial Intelligence (AI) entities, and can be seen from the end–user point of view as an intuitive representation of a butler, each of them with specific orders coded in their ECA rules. Thus, the natural process by which the user associates certain tasks (and the associated responsibility) to a particular person or assistant (e.g. gardeners, maids or butlers) is translated to the Ubiquitous Computing domain in the form of agents. Since responsibilities are distributed among agents, a malfunctioning rule is easier to locate, since the user knows which agent should be in charge of which particular task. Moreover, this modularization allows users to modify or eliminate those malfunctioning parts of the system without spreading the mistrust generated by the failure to the overall system.

Since agents can be activated or deactivated independently, new automation domains can be opened in the form of new agents. Thus, whole sets of preferences can be activated and deactivated according to the user's preferences.

The flexibility provided by a modularization mechanism is enhanced by the fact that no grouping constraint is forced. Contrary to other systems in which some kind of bundle is imposed (e.g. activity), our agents are only designed to be independent wrappers of rules, **the classification of the agent is up to the end–user**. Thus, depending on the users' preferences, agents will be designed to handle preferences related to activities (e.g. preferences for "watching a movie"), scenarios (e.g. preferences for when I am alone), control domains (e.g. lighting preferences), locations (e.g. preferences for the kitchen), devices (e.g. preferences for the kitchen switch), persons (e.g. Pablo's preferences) or any combination (e.g. Pablo's preferences for watching TV in the kitchen), among others.

This flexibility allows users to configure their environments according to their mental control structures, instead of forcing them to adapt their mental structures to the configuration mechanism for the environment.

In addition, **modularization empowers the scalability of the system**. Agents can be added to control new, previously uncontrolled scenarios. They can be removed when devices are removed or they can be duplicated when it is desired to replicate particular behaviors of other people's environments.

There is no accepted definition of what an agent is, and discussions abound on the subject [40]. Thus, it is important to define what our agents are and are not. Jennings et al. [68], in their definition of agents, highlighted three key concepts: *situatedness*, *autonomy*, and *flexibility*. We will use these concepts in order to define our agents.

Firstly, we share Jennings' et al. understanding of situatedness, meaning that "the agent receives sensory input from its environment and that it can perform actions which change the environment in some way". Secondly, in relation to Autonomy, we must stress that our agents are not autonomous in the classical sense of *autonomous agents*, but rather independent processing units, programmed entirely by end–users, with no negotiation or communication mechanisms among them. On the other hand, they show some soft autonomous features, such as a light learning regarding the performance of their rules, a passive notification mechanism about the learning and an *are you alive?* protocol. Therefore, we prefer to consider our agents to be independent (from each other and from the user, once programmed) rather than autonomous. Hence the stress we place throughout this thesis on the difference between autonomous and automatic systems. Finally, flexibility is defined by Jennings et al. as being *responsive* (i.e. "agents should perceive their environment and respond in a timely fashion to changes that occur in it"), *social* (i.e. "agents should be able to interact, when appropriate, [...] with humans in order to complete their own problem solving and to

help others with their activities") and *pro–active* (i.e. "agents should not simply act in response to their environment, they should be able to exhibit opportunistic, goal-directed behavior and take the initiative where appropriate"). While we share both the responsive and social features, we disagree with the pro–active property since, from the beginning, we designed the system to be automatic, not autonomous, in an attempt to create a system that is completely predictable for the user, that is, that it does simply what it has been told to do.

5.2 Agent anatomy

An agent is an independent inference engine defined through a file with three different sets of rules, *on load*, *on running* and *on finished*, defined in table 5.2

On load rules	A set of CA rules to be executed when the Agent starts.
On running rules	A set of ECA rules active while the Agent is running.
On finished rules	A set of CA rules to be executed when the Agent ends

Table 5.1: Parts of the Agent structure

These sets are structured in the file according to the following template:

```
{
   on_load_rules
}
{
   on_running_rules
}
{
   on_finished_rules
}
```

5.2.1 Internal indexing

The on–running ECA rules are internally indexed in three hash tables which index the rules through their triggers, conditions and actions, so when a change in a Blackboard element is received, retrieving the triggered rules, those whose conditions must be reevaluated and those whose actions may have recently affected the same element (and may be reeducated,) is a fast and straightforward process. A similar indexing is carried out with the on–load and on–finished CA rules (with the exception of the hash table for triggers) and for rules, indexing

their conditions and actions by their LHS and RHS elements for the same purpose (see Figure 5.1).

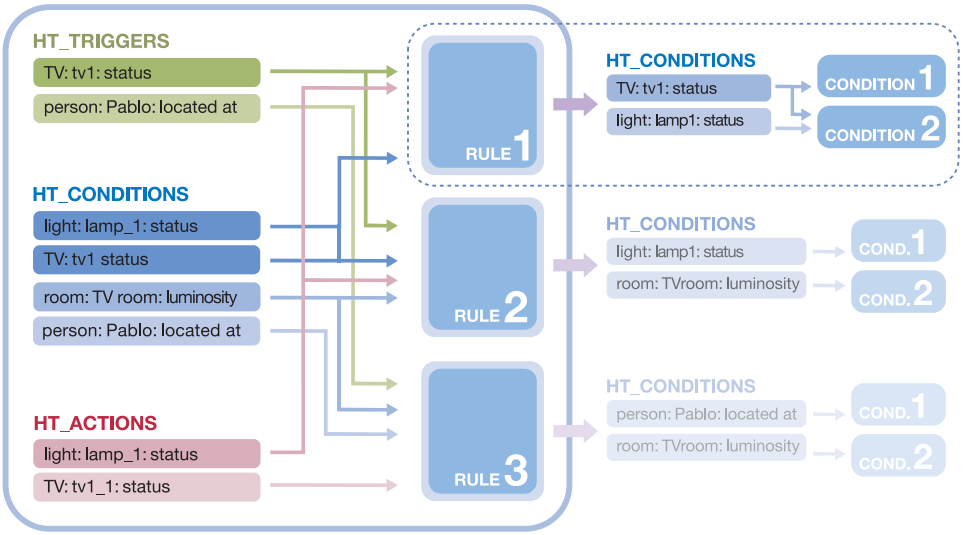


Figure 5.1: Internal structure of the agent with the hash table indexing of rules, conditions and actions

5.2.2 Blackboard representation

Each agent is represented in the blackboard as another part of the context in the same way a real butler would be represented as another person (but belonging to the Virtual World of figure 3.2 instead of to the Real World). The *agent* type has a property *status* and a compulsory relation *owner* linking it with the person or groups of persons for which it works (those who programmed it). Optionally, a property *task*, defining the goal the agent was programmed for, and a relation *location*, linking it to the physical bounds within which it acts, may help users to organize and manage their agents. On the other hand, their rules remain private, keeping the representation of the world function–free (see Figure 5.2). The only piece of internal function that is represented in the blackboard is the "affects" relation linking the agent with all the elements that can be modified by it (those present in the *actions* part of its ECA–rules). These properties and relations are defined in Table 5.2.2

This representation as another part of the world provides a simple mechanism for creating rules that act not only upon and on the state of the world, but also on the state of the agents (i.e. the rule engines), dynamically modifying the reasoning rules of the overall system according to context, or making decisions based on their state —e.g. *"When I enter the house activate my lighting preferences for the house" or "If all agents affecting the entrance door are deactivated, turn on*

Properties	Description	Compulsory
name	The name of the agent is actually the name assigned to the entity. To refer to the agent we will use the string *agent:<agent_name>*	yes
status	The main property of the agent. It can be set to *ACTIVE*, *LEARNING* or *INACTIVE*, meaning, respectively, that the agent is completely functional, is not taking any actions but continues listening and learning from what is happening in the environment, or that it is only listening for possible changes in its status.	yes
purpose	An optional property to allow users to categorize their agents according to their mental structures. It can take any string value	no
Relations	**Description**	**Compulsory**
owner	It links the agent to the person or group of persons for which it works.	yes
affects	This relation is automatically created between the agent and those elements present in the actions of its rules.	yes
location	An optional relation to an entity of the type *environment* (or any of its subtypes). It does not restrict the agent to act or respond only to elements in that environment, but allows users to categorize their agents in spatial terms.	no

Table 5.2: Properties and relations of the agent type

the alarm" (see Rule 5). This is an extremely useful point that allows end–users to control their preference sets through the same mechanism they use to control the rest of the environment.

R U alive?

It is important to note that the *status* property of the agent is not stored in the Blackboard; instead, every agent has a small server implemented. Through this server, agents accept requests from other processes to report their status. Thus, when a process asks the Blackboard for the status of a particular agent, the Blackboard asks the agent for the value before sending it back to the interested process. This mechanism works as an "are you alive?" mechanism, allowing for the identification of crashed agents by trying to retrieve their status.

Since agents are generated in the Blackboard on startup and removed on shutdown, this process is necessary for agents to be able to verify, on startup,

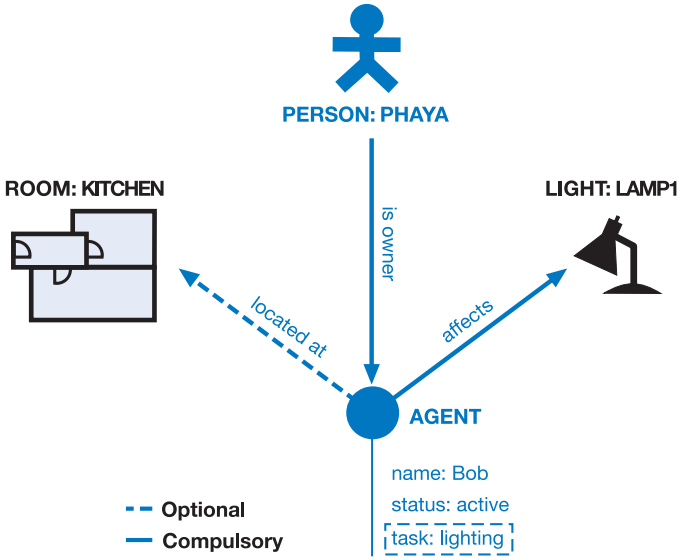


Figure 5.2: Representation of an agent, properties and relations in the Blackboard

Rule 5 Example of a rule codifying the behavior "If all agents affecting the entrance door are deactivated, turn on the alarm"

```
agent:*:status
 ::
  agent:*:affects = door:entrance_door
  &&
  agent:$1:status != ACTIVE
 =>
  alarm:main_alarm := ON
;
```

whether another agent with the same name already exists in the Blackboard (and notify the user to name the agent differently), or whether the agent represented in the Blackboard was supposed to be itself (crashed and restarted), so it can supersede it.

In addition, this mechanism can be used to periodically check the status of the agents, removing them from the Blackboard in case they are dead.

5.3 Execution model

The execution model specifies how the set of rules is treated at runtime [104], in other words, the rule execution process of a rule engine. Some properties arise in our system from the fact that there are different rule engines running

simultaneously. Firstly, rules are distributed among agents and each agent is a rule engine that is highly uncoupled from the remaining agents. Secondly, each timer is a new rule engine that is in some way associated with the agent and timers that created it. While the coupling is higher in the timer case, they still run in parallel threads and follow their independent execution model. No strict semaphores are available since no process is modeled in the Blackboard. Nevertheless, context variables are used in some cases to direct the execution flow; hence, while more independent than the timers, agents are represented in the Blackboard and their state can be changed through it, modifying the active rule engines of the system according to context. Timers, on the other hand, are just internal process of agents and cannot be modified but by their own rules.

Any form of rule engine (i.e. either timers or agents) shares the same execution model. This model has been designed to deal with the *passive* event detection mechanism of subscription provided by the Blackboard. A passive event detection mechanism was chosen over an active one (pull–model)so as to minimize the communication load between the Blackboard and the agents, as well as to improve the reasoning time performance of agents. Thus, conditions are updated as changes occur, so when a rule is triggered only the conditions involving the trigger have to be updated, avoiding the cost of querying the database at the crucial decision moment. Additionally, the shorter the time response must be, the more frequently the system has to query the database in a polling mechanism, overloading it with unnecessary requests. Besides, since our knowledge database is event–free, a passive mechanism has the additional benefit of implicitly describing events. Passive detection is used to update all condition values save those having *Wildcards* (see Section 4.1.2), which are resolved by querying the Blackboard in order to balance communication load between Blackboard and agents and the agents' cache size.

The execution begins with one of the five *callback* methods for structure updates available in the subscription mechanism: *Property changed*, *Entity removed*, *Relation removed*, *Entity added* and *Relation added*. The execution model can be divided into nine phases (see Figure 5.3):

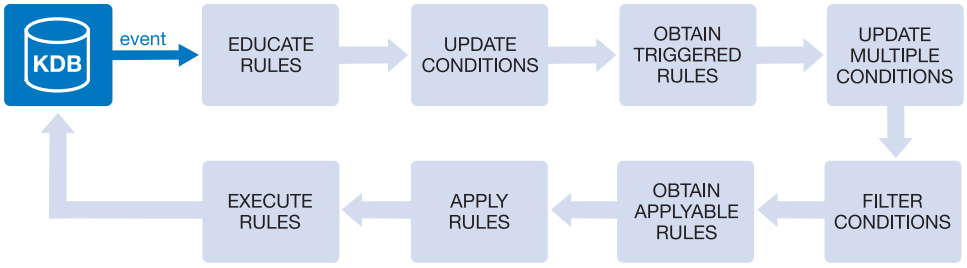


Figure 5.3: Forward directed execution model for ECA rules

- **Educate rules**: Educates previously executed actions according to the event received.

- **Update conditions**: Updates the truth values of any condition containing the changed element. Conditions can be accessed through hash tables within the rules, using as a hash key the elements involved in them. Rules are also hashed within the agent. If a condition changes its value, the number of true conditions in the rule is updated. Later, when checking whether all conditions for applying a rule are satisfied, this number is compared to the number of conditions in the rule.

- **Obtain triggered rules**: Since events are just a conjunction of raw events, obtaining the triggered rules is as simple as retrieving from a hash table all the entries corresponding to the same hash key (i.e. the changing element), e.g. a rule with the triggers $light : lamp\_1 : status \;||\; person : mherranz : locatedat$ would be indexed twice in the hash table for the $light : lamp\_1 : status$ and $person : mherranz : locatedat$ hash keys.

- **Update multiple conditions**: Updates the values of all conditions referring to a Wildcard in the trigger part that can be matched with the changed element. For example, if $light : lamp\_1 : status$ has changed, it updates the value of the condition in the rule $light : * : status :: light : \$0 : status = 1$ as if it were the rule $light : lamp\_1 : status :: light : lamp\_1 : status = 1$

- **Filter conditions**: It updates the values of all conditions of triggered rules with Wildcards, updating their values from left to right.

- **Obtain applicable rules**: For all triggered rules, if the number of conditions and true conditions are equal, the rule is tagged for execution.

- **Apply rules**: The actions of tagged rules are translated into structure updates to send to the blackboard. Actions belonging to the same rule are codified as transactions.

- **Execute rules**: All structure updates are sent to the Blackboard.

This process is repeated with every notification sent from the Blackboard. Notifications are queued as they arrive and processed sequentially. Timers, on the other hand, have their own subscription mechanism and are executed in parallel with the agent that created them, having their own notification queues. Thus, in the same agent, events can be processed sequentially or in parallel, according to the programmer's needs.

5.3.1 Explanation

The results of the execution phases are shown and recorded in a log. ECA–rules being human–readable structures, this log serves as an explanation mechanism of the agent's internal reasoning process and can be used for debugging purposes. This, as stated in Chapter 3, is one of the solutions designed to improve competence: since programs are developed and maintained by non–professional programmers, a natural debugging (i.e. explanation) mechanism must be provided. In addition, making available the internal reasoning process of agents also engenders trust.

By way of explanation, each agent provides a log of its reasoning process in real time, allowing the user to choose between a basic mode and a verbose mode (in which information is provided in more detail) (see Table 5.3.1). In addition, agents can list their rule sets on request, showing their conditions with their respective truth values. This makes it possible to spot an incoherence between the real state of the environment and that perceived by the agent.

Exec. state	Plain mode	Verbose mode
Event	Shows the event received from the Blackboard.	Shows the event received from the Blackboard.
Education	Shows the actions that have been educated (if any).	Shows the actions that have been educated (if any) and the list of actions that remain to be educated.
Conditions update		Shows which conditions of what rules have been updated
Triggered rules		Shows the list of triggered rules
Filter multiple conditions		Shows the filtered sets of conditions
Applicable rules		Shows the list of rules that are to be applied
Executed rules	Shows the list of actions that are sent to the Blackboard	Shows the list of actions that are sent to the Blackboard

Table 5.3: Structure for the agent explanation log

5.3.2 Learning

In an attempt to empower competence in an end–user programmed system, an additional learning mechanism has been implemented. Other Ubiquitous Computing systems, such as the MavHome [23], focus on environments that try to learn from users' behaviors in order to anticipate their actions. Ours, on the other hand, aims to be as predictable as possible, not automating anything the user has not tried to automate explicitly. Thus, the purpose of our learning mechanism, far from anticipating the user, is to locate and possibly solve errors in the user's programs.

ECA–rules, as an explicit desire for intelligence, provide an implicitly open door for learning. E.g. once the lights have been programmed to be adjusted according to people getting in an out of the room, users will not be surprised if the lights go on and off as people gets in an out of the room so automatic learning upon that basis will only improve the competence of an expected behavior. This predictability is especially important in personal environments, such as a home.

To do so, every time an action is executed by an agent, it is tagged for education. When a context update is received, all the actions not responsible for that change are educated and untagged for education.

Education is based on reinforcement learning. Each time a rule is executed an education time interval is opened for reinforcement. If within this interval a context update contradicts what the action commanded, the agent understands that it was unwanted and decreases the *confidence factor* (from now on CF) of the action (see Section 4.1.3); otherwise, it increases it.

If the CF drops below a certain value (i.e. the deactivation threshold), the agent may, according to how the user configured it, automatically disable the action or notify the user of the possible malfunctioning part.

Even when an action has been disabled the reinforcement process continues, using what the action may have done (had it been enabled) to possibly increase its CF (see Figure 5.4).

If the CF reaches a reactivation threshold, the action enters a reactivation process similar to (but with the opposite effect of) the deactivation one 5.4).

The behavior of CFs over time gives further information about the action's correctness. Thus, while constantly increased or decreased CFs are signs of correct and incorrect actions, respectively, a sequentially increased and decreased CF corresponds to an action associated with an ambiguous context. This context has to be disambiguated by changing/adding/deleting some conditions (or events) of the rule. Comparing the inflexion points in the CF evolution curve with the Blackboard values at those times may provide a hint about what variables must be added to the rule. Physical location may help to reduce the context to analyze, since most preferences work on local variables.

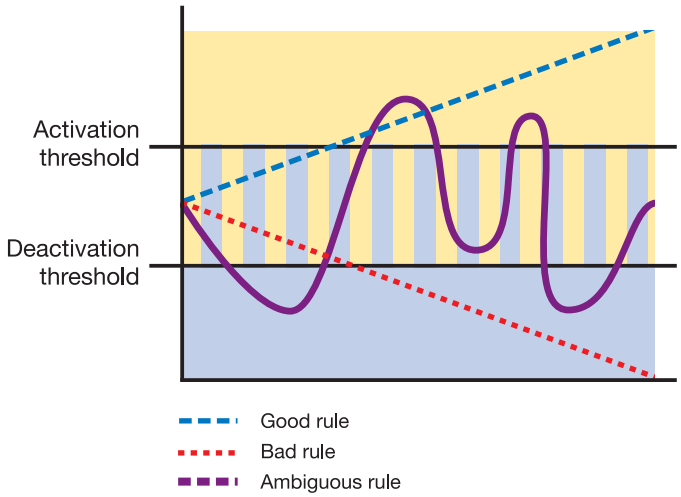


Figure 5.4: Evolution of confidence factors over time in relation to action correctness

5.4 Managing hierarchies

When multiple users share the same space, as is normally the case, their diversity of preferences is reflected in a multitude of context–aware applications running over the same environment. In traditional environments, these preferences are prioritized according to some ordering mechanism by the inhabitants. Analogously, since context–aware applications are automated user preferences, they must have a similar mechanism through which to apply the same ordering (applied in traditional environments by the inhabitants themselves) if they are to coexist in the same environment.

Hierarchies are the natural social structures for establishing an order of preference but, linked as they are to the social group that created them, they are multiple and dynamic and their complexity reflects the complexity of the social group they rule. In addition, every social group has its own ways of generating and accepting their hierarchies.

Managing preferences is closely related to the problem of creating hierarchies. Multiple users inhabiting the same space make the interaction dependent on the remaining users and their preferences. The problem of co–existence is as old as mankind itself, and any system for home automation ignoring this aspect would fail at some point.

The generation and acceptance of the social hierarchy is probably one of its most important aspects. When a conflict arises, the social group in conflict creates a hierarchy to solve it. The rules defined when delineating the hierarchy are intended to keep the same conflict from arising in the future. To achieve this, every member of the society must believe that the established rules are fair. That

is, that the result of the previous fight would be repeated if fought again. What "weapons" are used in the fight and who and wins it, and how, are factors that depend on the idiosyncrasies of the particular social group, the chronological moment and the particular context of the fight since **every society has its own ways of generating and accepting its hierarchies**. In this sense, the acceptance of a hierarchy by the members of a society is closely related to the fact that it is established according to their own values and ruling methods.

This way of looking at humans as the only generators/acceptors of hierarchies is closely related to the very essence of this work. As we look to enrich the control users' experience over their environments —in contrast to other systems where the focus is on autonomous artificial intelligence— we believe, as did Taylor [114], that the environment's intelligence might be judged by how it enables us to be aware of the world and act on it. Thus, the environment's "intelligence" is just a complement to our own, true Intelligence and main engine of the environment.

Going back to indirect control over intelligent environments, the automatic decision–making mechanism aims to be just an extension of the user's control. Decisions are automatic in the sense that they do not need direct human intervention. It is important to point out, however, that behind any action or command in the environment there is a human element. That is, automatic decisions are human decisions –indirectly– and, as such, they are governed by the same hierarchies governing users' daily life.

Given that every action produced by an agent in the environment is —indirectly— produced by a human, we believe that **neither the system automatically nor a third human party, such as an engineer, should specify the hierarchies** governing a social group. They should instead allow each social group to express its own. This apparently naive principle poses a profound problem when designing a solution, since any fixed structure will interfere with the possibly different natural structure of the user. Thus, the structure must be flexible enough to adapt to the natural structures and organizations of social groups. We propose two social factors for analyzing these structures and organizations : **purpose** and **complexity**

5.4.1 Purpose

Since each type of conflict produces a different hierarchy domain, we find that the *purpose* for those hierarchies that are built is varied and entangled. Social hierarchies (i.e. *who goes first?*) and task hierarchies (i.e. *What goes first?*) are just two examples of a potential clash: What happens if there is a conflict between a high–priority person doing a low–priority task and a low–priority person doing a high–priority task? A new hierarchy (or an exception) will be born: a hierarchy of hierarchies, so to speak.

Some systems have handled this problem through classifying, organizing and

structuring. Kakas et al. [69], for example, provide each agent of a multi–agent system with two types of priority rules: *Role priorities* and *Context priorities*. Role rules prioritize according to the *role* of the parts involved (some sort of social hierarchy), while context rules prioritize role rules according to some extra context (some sort of task hierarchy). Additionally, each agent is supplied with a *motivation* structure, a hierarchy over the goals it pursues. These goals are defined according to Maslow's needs categorization [79]. Finally, an additional hierarchy is specified to define the agent's "personality" (i.e. its decision policy on needs to accomplish the goals of its motivation).

While this kind of structuring captures many of the flavors of human behavior, it presents some problems when applied to personal environments. First, it needs a professional (expert in Maslow's theory) to program the system, and a clear a priori classification of the users' goals, roles and tasks. These engineering solutions, while perfectly valid for domains such as business automation, should not be applied to home environments in which a programming third party, an overly rigid categorization or the need of a deep a priori definition breaks some of Davidoff's principles for smart home control, such as "allow an organic evolution", "easily construct/modify behaviors", "understand improvisation" or "participate in the construction of family identity"[29]. Thus, hierarchies should not be constrained to specific purposes or domains (e.g. roles or goals), nor presume a knowledge of complex concepts (e.g. Maslow theory).

5.4.2 Complexity

Secondly, as a social structure, we considered *complexity* the second factor of interest while analyzing hierarchies. How do we allow for the various degrees of complexity of different social groups? How can we measure those different degrees of complexity? Y. Bar–Yam's [8] *interdependence* and *scale* concepts to measure complexity are quite useful in this undertaking.

Interdependence describes the effects a part has over the rest of the system: "If we take one part of the system away, how will this part be affected, and how will the others be affected?" Not only will different systems suffer different effects (Bar–Yam exemplifies it by removing a part from a metal, a plant and an animal and analyzing the part and the crippled system: How well does the part represent the system? How well does the system behave without the part? Does it matter which part is removed?) but, more importantly, interdependences are not equally easy to characterize in every system. In fact, as Bar–Yam points out, "natural system components are typically interdependent in ways that are not readily obvious" (What will happen if an adenine is removed from a DNA chain? or a company from the stock market?).

Scale, on the other hand, refers to the different degrees of complexity a system acquires depending on how close or far removed the observer is. That is, Bar–

Yam considers complexity as a subjective measure that depends not only on the system, but also on the distance between the observer and the system. To exemplify this he considers our planet and how it is seen as a single point moving predictably along its orbit when seen from far away and as an incredibly complex system (with a plethora of unpredictable variables such as movements in the atmosphere, oceans, plants or human beings) when viewed in greater detail. The important point of scale is not only that it changes the perceived complexity of the system, but also that this change characterizes the system: "the variation of complexity as scale varies can reveal important properties of a system".

Basically, the study of complexity is not focused on the study of the parts of a system, but on the relations between them. Interdependence and scale, as a characterization of the nature of these relations, are helpful variables to characterize, classify and describe complex systems. Thus, as highlighted by Bar–Yam, while both a designed system such as a microprocessor, and a spontaneous one such as the global economy, have millions of components, sharp differences can be found between them in terms of how well the system can be understood and how sensitive the system is to improvements or changes. On the one hand, a microprocessor has been designed and tested, therefore it is more understandable but, conversely, its growth depends on a new design and testing process. On the other hand, the global economy –as a spontaneous system– is not fully understood nor controlled by anybody but is more robust and adaptive to changes in the environment and, in the overall scale, can dominate disturbances and changes in its subcomponents, to name just some of the consequences of the idiosyncrasies of their complexity.

Social networks range from the meticulously planned, as in military hierarchies, to highly spontaneous, as in the Internet. Some have their complexity spread among the structure (e.g. the Ford T production chain), while others condense it in specific parts (e.g. a diamond cutting business). When looking at computer solutions for managing social structures, the different degrees of interdependence and scale they allow are easy to see and measure. These degrees are, in turn, a measure of the degrees of complexity that can be achieved with it.

It is of vital importance that the diversity of social structures be considered when designing a control mechanism to guide and manage said structures. Thus, this mechanism, in addition to allowing the creation of strict and centralized hierarchies, must provide more flexible mechanisms to replicate the natural coordination structures of human societies, despite the loss of control and understanding brought on by spontaneous structures when compared to planned ones (e.g. ignoring the a priori degree of interdependence implies ignoring the effect of removing a part from the overall system). In order to measure how flexible a system is (i.e. how many degrees of complexity it can replicate), we can analyze the different degrees of interdependence and scale it supports.

In conclusion, if we want to replicate different levels of interdependence we need to provide a decentralized mechanism for hierarchy management. A mechanism of this kind, in which there is not necessarily a central agent coordinating the processes, does not prevent the creation of centralized hierarchies, but it does not impose them either. To enable different degrees of scale, on the other hand, we need to provide different levels of programming. How many layers the solution is structured into and how much complexity each of them holds will be up to the users and depend on the problem they intend to solve.

5.4.3 Hierarchies – structure and definition: Multilayer filter

Now, when developing a mechanism to manage hierarchies, besides the underlying structure, there are two more issues to take into account. The first refers to the problem of *Who is in charge of defining the hierarchies*. The second, on the other hand, deals with the *What kind of conflicts is it designed to solve* issue, since not every kind of conflict is of the same nature and, consequently, is not directed by the same control structures.

Once the decision is made not to impose any underlying structure to users for creating their hierarchies, we must define the means by which users can express their hierarchies in the Ubiquitous Computing. Doing so requires summarizing some of the most important modularization properties of the system:

- A preference or task may include more than one action. Thus, decision rules may be grouped into sets, depending on the preference or task they belong to. Those sets are the agents explained in Section 5.

- Agents may be activated or deactivated according to context and they must be understood as another piece of context. In other words, agents are present in the blackboard layer [60] and may be activated/deactivated through it.

- Agents must *belong to* a user or group of users, their actions being an extension of their owners'. Additionally they are related to the elements of the world they *affect* and may be tagged with the *activity/purpose* they are designed for and the *location* in which they work (see Section 5.2.2).

These properties have been established to provide the necessary modularization and structure to the indirect control mechanism as a response to the two keystones posed by P. Maes [78] in the design of software agents: *competence* and *trust*. Firstly, it favors competence by allowing the creation of complex tasks –composed of many preferences– but treatable, form the activation/deactivation point of view, as a whole. Secondly, since every agent is associated with a person

or group of persons, a location and an activity, the activation/deactivation process can be conducted in the familiar terms of the natural environment's social and activities hierarchies, thus favoring trust.

With these properties in mind, and according to the nature of conflicts, we have implemented a multi–layer structure to solve conflicts (see Figure 5.5). This structure acts as a series of filters between the actions commanded by ECA–rules and their effect on the world. Thus, an action represent an individual's (or individuals') desire, while the hierarchies (codified in the multi–layer structure) represent the social conventions or global interest. For a desire to change the world, it must go through the social conventions or hierarchies. These hierarchies can agree with the desire and let it pass through, cancel it or alter it in some way, as we will explain.

This structure uses the AmILab layer structure (see Figure 3.2) to deal with conflicts of different natures.

Conflicts are classified into two categories: **ownership** and **collisions**. In addition, we distinguish between two kinds of collisions: *context–dependent collisions* and *context–independent collisions*, according to whether the only important factors to solve the conflict are the elements colliding and the object of collision (the commanders and the entity), or there is any other relevant element in the resolution.

Ownership conflicts and context–independent collisions, as we will show, can be solved in the *Privacy layer* and the *Blackboard* of the Context layer. Context–dependent collisions, on the other hand, must be solved in the interaction layer, for which we propose the use of the same rule–based agent mechanism mentioned in Section 5, allowing end–users to **use the same programming structure they use to program their preferences to program the hierarchies that control those preferences**. Thus, control is unified in the interaction layer: i.e. everything, from the elements of the environment, the hierarchical policies, the task flows or even the privacy settings, can be programmed by the user in the same manner, creating a sound controlling experience in which the users perceive their environment, in terms of programming, with no artificial bounds.

Filtering in the Context layer

Analyzing the layers from the environment to the user (see Figure 5.5), the Blackboard layer, where every element is represented, provides a priority queue for each element [62]. This queue is governed by a policy and acts as the default policy to apply in case of collision (i.e. if despite the rest of the layers a collision reaches this point, the default policy is applied). The policy defines a time interval for considering two orders as colliding and a function to apply taking into account the actors in conflict and the element itself. This function gives a priority value to each order, with the highest value taking precedence. Considering a door, an

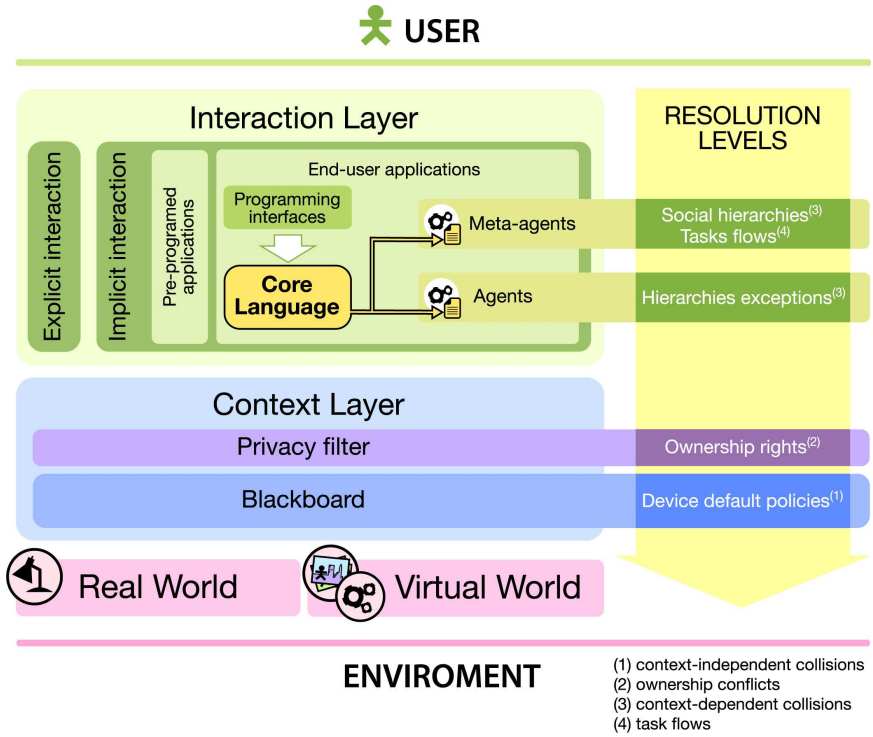


Figure 5.5: Amilab layers from the conflict resolution point of view

example would be the following:

$$P = is_{person} * 0.5 + is_{application} * 0.3 + is_{security} * 0.4$$

P	*Actor*
0.9	Security person
0.7	Security application
0.5	Person
0.3	Application

Where is_{person}, $is_{application}$ and $is_{security}$ are binary variables with value 1 if the commander is a person, an application or part of the security staff, respectively, and 0 otherwise. As an example, a command from a non member of security trying to open the door will be assigned a priority value of $1 * 0.5 + 0 * 0.3 + 0 * 0.4 = 0.5$, while a security application trying to close it at the same time will receive a $0 * 0.5 + 1 * 0.3 + 1 * 0.4 = 0.7$. Consequently, the door will close.

Secondly, the Privacy layer allows users to establish access rights to the elements they own [36]. Besides being used as a privacy filter, it helps in constructing hierarchies where the owner is the only relevant factor in the policy (i.e. users

freely control the access to their elements). Policies can be established for individuals, groups of individuals or following a "fair trade metaphor" in which users can only access information (or control elements) of others to the same degree they allow others to access (or control) their own. Privacy information is also represented in the Blackboard (like any other element is); therefore, as we will see, it can be changed depending on context through the interaction layer.

Filtering in the Logical layer

Finally, through the Interaction layer, users can create structures to control the environment that are, in turn, represented as part of the environment. In this way they can create agents whose rules control the status of other agents (instead of a physical element of the environment). We will refer to these agents as meta–agents. Similarly, a rule can be used to set the privacy preferences. This mechanism for controlling indirect control structures allows for the creation of hierarchies in as many levels as desired. The complexity of the system —interdependence and scale— is up to the inhabitants and their natural hierarchies. Interdependence is easy to see (while not always easy to understand) in the Blackboard as the graph created by all the *affects* relations. Depending on the scenario this graph will range from pyramidal structures to unconnected graphs or entangled networks, with a person(s) behind each agent, an element of the environment in each leaf of the graph and, in between, a complex structure of conditions that, as a whole, governs the overall automatic behavior of the environment (see Figure 5.6). Scale, on the other hand, can be appreciated in the different levels in which hierarchies can be expressed.

To illustrate this with an example, let us consider two users sharing a house. User A prefers the light level to be low while user B prefers it high. In this situation, they can control their preferences through a single agent (associated with both of them) in which three rules codify their preference: "if user A is in the house but not B, when watching TV, set the light level to low", "if user B is in the house but not A, when watching TV, set the light level to high" and "if both A and B are in the house, when watching TV, set the light level to medium" (see Rule 6).

Conversely, they can have an agent for each, codifying their personal preferences (see Rule 7), another shared agent codifying their mutual preferences (i.e. what they want when they are together) (see Rule 8) and a meta–agent deactivating their personal agents and activating the shared one when both of them are in the house and vice versa (see Rule 9).

Or, finally, they can do without agents and establish a default policy in the Blackboard (since no other factor but themselves and the light is present in the conflict) to establish *the average* as the desired value for the light when a conflict arises.

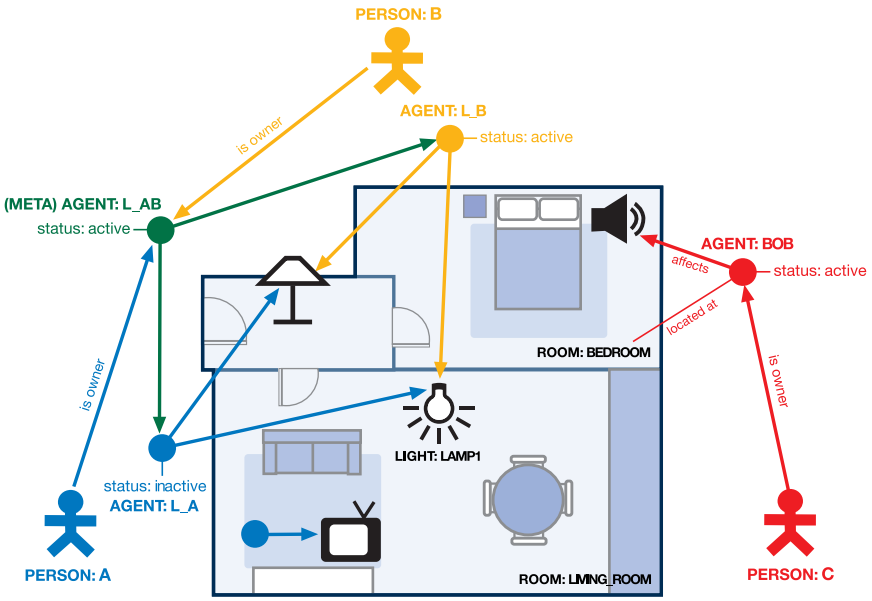


Figure 5.6: An example of interdependence in the graph created by the connections between people and their agents (*is_owner*), and the agents with the objects they affect (*affects*).

While codifying the same behavior, the three approaches present a different interdependence and scale and they will be preferred over the others according to the idiosyncrasy of the social group. Thus, the former will be more frequent in situations in which most of the preferences are shared (e.g. a couple sharing a house), the second when each individual normally decides alone and some coordinating mechanism is required (e.g. student roommates), while the third one is more natural to sporadic environments in which personal preferences are secondary (e.g. a laboratory hallway). These structures have been observed in the three real environments in which the system is deployed: a simulated living–room in the AmILab laboratory (Autonomous University of Madrid, Spain), a simulated security chamber at Indra's facilities (Madrid) and an intelligent classroom in the Itechcalli laboratory (Zacatecas, Mexico).

Cooperation between Logical and Context layers

Finally, we would like to briefly point out some synergies between layers. The first one has already been mentioned and has to do with the possibilities provided by the indirect control mechanism to change the privacy settings according to context. The second one, conversely, has to do with how owner rights can be automatically used to establish rights over agents.

While an element represents an object (either physical or virtual) for which ownership rights are clearly defined, an agent represents a set of preferences

Rule 6 Example of hierarchy to set the light level when watching TV, according to whether user A and B are in the room, codified in a single agent with three rules

Agent: *light* **Owner:** *person:A, person:B*

```
tv:tv_1:status
 ::
  tv:tv_1:status = ON
  &&
  person:A:locatedat = tv:tv_1:locatedat
  &&
  person:B:locatedat != tv:tv_1:locatedat
 =>
  light:dim_light:value := LOW
;

tv:tv_1:status
 ::
  tv:tv_1:status = ON
  &&
  person:A:locatedat != tv:tv_1:locatedat
  &&
  person:B:locatedat = tv:tv_1:locatedat
 =>
  light:dim_light:value := HIGH
;

tv:tv_1:status
 ::
  tv:tv_1:status = ON
  &&
  person:A:locatedat = tv:tv_1:locatedat
  &&
  person:B:locatedat = tv:tv_1:locatedat
 =>
  light:dim_light:value := MEDIUM
;
```

or desires, so the question *"Who has the right to affect others' preferences?"* becomes non–trivial. It may seem that preferences are exclusive the property of their owner, but if they are preferences over someone else's or shared objects, the problem becomes more complicated.

Forcing users to establish privileges not only over their objects but also over their preferences can be fairly intrusive. In fact, avoiding such processes in rela-

Rule 7 Example of two different agents holding the preferences of user A and B respectively for the light when watching TV

Agent: *lightA* **Owner:** *person:A*

```
tv:tv_1:status
 ::
  tv:tv_1:status = ON
 =>
  light:dim_light:value := LOW
;
```

Agent: *lightB* **Owner:** *person:B*

```
tv:tv_1:status
 ::
  tv:tv_1:status = ON
 =>
  light:dim_light:value := HIGH
;
```

Rule 8 Example of an agent holding the agreed-on preferences of A and B when watching TV together

Agent: *lightAB* **Owner:** *person:A, person:B*

```
tv:tv_1:status
 ::
  tv:tv_1:status = ON
 =>
  light:dim_light:value := HIGH
;
```

tion to objects (and information) is the main goal of the Privacy layer and the "Fair trade" policy for privacy management it provides [36].

While we strongly believe that conflicts must be solved at the human level, information technologies can ease the process of finding them before they occur. In this sense, the ownership and privileges information of the Blackboard and Privacy layers may be used to find potential conflicts in meta–agents affecting other agents.

Three factors are of main interest for this process: what elements are affected by the agent, who the owner (or who has privileges) of these elements is and who the owner of the meta–agent is. The basic idea is that if the owners of the meta–agent have fewer privileges over the elements controlled by the agent than the owners of the agent, a potential conflict arises.

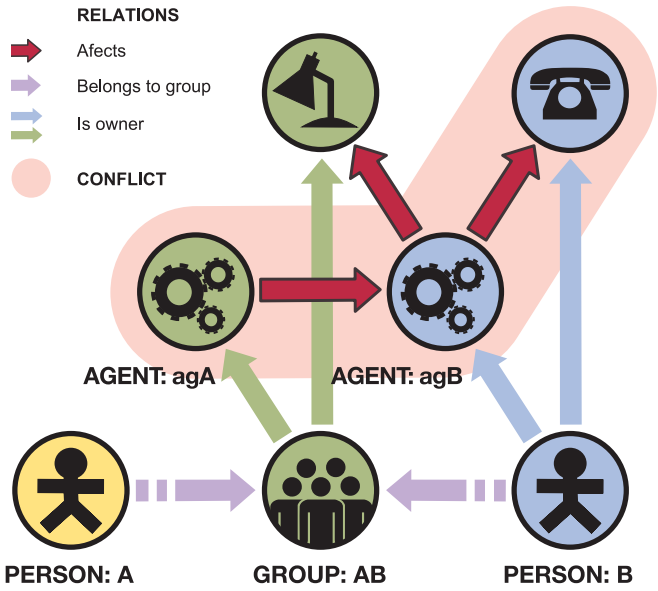


Figure 5.7: Extending owner information to automatically establish meta–agent permissions. The example shows a conflict found automatically when a meta–agent (belonging to the green group) tries to control an agent affecting an object not belonging to all members of the group

As an example, consider the meta–agent $agAB$ and the agent agB in Figure 5.7. Agent agB has been created by user A and has some rules to modify the lamp and the telephone. While the lamp is owned by the group AB, composed of users A and B, the telephone is the exclusive property of user A. Now, when the meta–agent $agAB$, created by group AB, tries to modify the state of agent agB, the preferences of user B over the telephone may be controlling A's preferences over the telephone. Since the telephone is exclusively owned by A, a human conflict may arise at some point (i.e. some person is trying to decide about my preferences over my objects).

Once a potential conflict is identified, different strategies may be applied. The most direct entail notifying the users of the potential conflict (leaving any solution to their criteria) or splitting agent agB into two agents, agB^1 and agB^2, agB^1 comprising only the rules affecting those elements owned solely by A and B (i.e. the lamp) and agB^2 comprising the rest (i.e. the telephone). Then, $agAB$'s rules affecting agB can be changed to affect agB^1 instead.

Rule 9 Example of a meta–agent coordinating agents according to context. If A and B are in the TV room, their respective agents are deactivated and the common agent is activated. If either of them is alone in the room, his/her agent is activated. If either of them leaves the room his/her agent is deactivated

Agent: *coordinationAB* **Owner:** *person:A, person:B*

```
person:A:locatedat || person:B:locatedat
 ::
  person:A:locatedat = tv:tv_1:locatedat  &&
  person:B:locatedat = tv:tv_1:locatedat
 =>
  agent:lightA:status := INACTIVE  &&
  agent:lightB:status := INACTIVE  &&
  agent:lightAB:status := ACTIVE
;

person:A:locatedat | person:B:locatedat
 ::
  person:A:locatedat = tv:tv_1:locatedat  &&
  person:B:locatedat != tv:tv_1:locatedat
 =>
  agent:lightA:status := ACTIVE
;

person:A:locatedat | person:B:locatedat
 ::
  person:A:locatedat != tv:tv_1:locatedat  &&
  person:B:locatedat = tv:tv_1:locatedat
 =>
  agent:lightB:status := ACTIVE
;

person:A:locatedat
 ::
  person:A:locatedat != tv:tv_1:locatedat
 =>
  agent:lightA:status := INACTIVE
;

person:B:locatedat
 ::
  person:B:locatedat != tv:tv_1:locatedat
 =>
  agent:lightB:status := INACTIVE
;
```

Chapter 6

Demonstrators

The system was intended from the beginning to be a working system with which to design real solutions to context–aware problems. It is currently running in three different laboratories and has been tested in combination with other state–of–the–art Ubiquitous Computing technologies. In addition, we have developed a GUI interface to ease the creation of rules and an early prototype of an end–user GUI.

6.1 Applied Environments

Each of the three laboratories in which the system is deployed was designed to study the possibilities of Ambient Intelligence in different types of environments. In the next subsections we will present some of the most relevant characteristics of these laboratories.

6.1.1 AmIlab

The Ambient Intelligence Laboratory at the Universidad Autónoma of Madrid is the oldest. Dating from 1999, its main focus is on applying Ubiquitous Computing technologies to living spaces. With an intelligent living room (see Figure 6.1) and a small working area, it holds up to 10 inhabitants in a variety of natural scenarios such as tea times, birthday celebrations, work meetings, movie projections or work hours.

In addition to the variety of scenarios, this laboratory is used on a daily basis by students and researches to develop and test other ambient intelligence technologies such as direct control mechanisms, multitouch surfaces, intelligent objects and privacy–aware applications. This presents two additional scenarios in which to use and test the indirect control mechanism, in addition to the above goal

Figure 6.1: A view of the laboratory B-403 living room at the Universidad Autónoma of Madrid, Spain

of studying its application in personal environments: work places and research places.

Personal spaces

As stated by Kidd et al [72], personal spaces are characterized by being "free–choice" environments, thus the amount of activities, scenarios and, consequently, preferences presented in them can be extremely varied. Most scenarios consist of a limited set of preferences, conforming an overall view of multiple domains with few specifications each. In this sense, the simulated AmIlab living room has been populated with a diversity of agents, developed over time as new preferences appear in what can be considered to follow an "organic evolution". Extending the behaviors of the system as preferences arise, in contrast to a designed model, allowed us to observe a singular trend in the rules. Thereby, Wildcards were hardly used, adding for example a new access rule for each person joining the environment instead of a general rule of the type "when a person...", like the one shown in Rule 11. In addition, many agents were created, with different purposes such as adjusting the lights when watching TV, showing environmental information on a particular display, greeting upon entrance, or controlling the TV from the couch. Agents had an average of 6 rules.

Since the simulated living room is somewhat integrated with the working area of the lab, certain other agents were created to support daily activities in it. Even though a laboratory has some physical similarities with a personal space,

it is shared by many more people with weaker relationships (than those of a family). In addition, some devices, such as a shower, are present while others, such as personal computers, can be found in greater numbers. In this sense, while some of the agents built in the environment to support daily living cannot be considered to be developed for an average home, given the low population of the lab (ranging from 2 to 10 inhabitants, depending on the time) and the home–like structure of most of it, they can be considered to be built to meet some of the requirements of a particular personal space.

In our case, as an example, our lab had many lights and lamps but only two switches (up and down). At the beginning, switches were associated with the ceiling light, while the rest of the lights could be controlled through the direct control GUI [2][50] and oral interfaces [86]. Since these interfaces were not always running, some of us found some nights —when leaving late after a day of joy and numerous accomplishments— that, before closing the door and going home, we had to restart a computer to turn off some of the lights that were still on (the same problem can be found in a large home when, after putting on our pajamas and getting ready to go to bed, we realize that we have left the garage light on, two floors down). To solve this problem, we enhanced the agent to control the actions of the switches to transform them into context–aware switches. Thus, when pressing the down switch, if the main light was still on, it was turned off. If the main light was off but the second light was on, then the second light was turned off and so on. Then, we added the same behaviors to the upper switch but to turn the lights on (preserving the same order i.e. main light first, then second light, and so on) and added some intermediate states to the dimmable lights so they could go from on to off through a medium state (see Rule 10). In this way we could control every light from a single place: a traditional switch. Anyone familiar with the Hanoi towers puzzle will realize that any lighting combination can be achieved in the room with a succession of pressing the up and down switches. Nevertheless, for a visitor, unaware of these capabilities, the switch continues to work as expected: the up switch turning on the main light and the down switch turning it off. The writer of these lines used to press four ups followed by two downs as he entered the room: half a second of what became an automatic movement to set the two dimmable lights to their medium levels with no other lights turned on.

Working spaces

Even though AmIlab focuses on the study of Ambient Intelligence applied to personal environments, it is in itself a work place. Having deployed the agent–rules mechanism as a working application, it has been used not only to design applications that could be useful in a personal environment, but also to support the daily activities of the laboratory. Thus, this daily experience has been useful

Rule 10 Example of rules for controlling a light and a dimmable lamp in the environment with only one switch

```
device:switchUp:value ::
  light:lamp_1:status = OFF =>
    light:lamp_1:status := ON
;

device:switchUp:value ::
  light:lamp_1:status = ON &&
  dimmablelight:lampv1:value < 50 =>
    dimmablelight:lampv1:value := 50
;

device:switchUp:value ::
  light:lamp_1:status = ON &&
  dimmablelight:lampv1:value > 50 &&
  dimmablelight:lampv1:value < 100 =>
    dimmablelight:lampv1:value := 100
;

device:switchDown:value ::
  light:lamp_1:status = ON =>
    light:lamp_1:status := OFF
;

device:switchDown:value ::
  light:lamp_1:status = OFF &&
  dimmablelight:lampv1:value > 50 =>
    dimmablelight:lampv1:value := 50
;

device:switchDown:value ::
  light:lamp_1:status = OFF &&
  dimmablelight:lampv1:value < 50 &&
  dimmablelight:lampv1:value > 0 =>
    dimmablelight:lampv1:value := 0
;
```

for testing its robustness as well as its adaptation to a non–fictional working space scenario.

One of the most characteristic examples of this kind of environment can be seen in the access control agent. Contrary to personal environments, the members

of a work place change more frequently, therefore access policies have to adapt more often. In addition, the social organization is more fixed and policies and preferences are consciously designed. In this case, access rights were controlled through a generic rule (see Rule 11) taking into account the membership of the research group.

Rule 11 Example of rules for RFID door reader to grant access to the laboratory

```
cardreader:door:card ::
  person:*:card = cardreader:door:card &&
  person:$0:belongsto = group:amilab &&
  door:main_door:status := CLOSE =>
    lock:door_B403:status := OPEN &&
;
```

Research spaces

Finally, being a research laboratory, some of the technologies being studied had special context needs for which the environment had no solution. This is the case of a privacy–aware application that, in addition to the privacy configuration, also needed to know location information. In this sense, the rule's mechanism was used to provide location information. This information, far from reliable in a deployed environment, allowed us to test our technologies in otherwise impossible scenarios. Some rules of the location agent, using the PCs and RFID card reader to infer location information, can be found in Rule 12. In addition, the rules were used to easily develop proofs of concept to test context–aware ideas such as a switch that, depending on who is pressing it, acts over one element or another. As an example, configuring the switch to turn on and off the TV (instead of the lights) when a certain RFID card was placed in a nearby RFID reader was done by adding a single meta–agent, with only 4 rules (2 to control the switch when the card was in the RFID and 2 to deactivate/activate the former switch agent when the card was inserted/removed). This meta–agent was added without having to modify any other system agent, not even the one controlling the switch, and enabled us to run some trials with a visitor to see her reaction to such an idea. □

In summary, the agent system is used daily at AmIlab to automate tasks such as access control, lighting preferences, message delivery, device enhancement (such as context–aware switches), or extracting high–level information (such as location) from low–level information. Its set of rules and agents have been created progressively over the last three years, as preferences arose from different users, to deal with different day–to–day living/working/research problems.

Rule 12 Example of rules using PCs and the RFID door reader to infer location

```
cardreader:door:card ::
  person:*:card = cardreader:door:card &&
  person:$0:locatedat != room:lab_b403 =>
    person:$0:locatedat -> room:lab_b403
;

cardreader:door:card ::
  person:*:card = cardreader:door:card &&
  person:$0:locatedat = room:lab_b403 =>
    person:$0:locatedat -< room:lab_b403
;

pc:*:status ::
  pc:$0:status = BUSY &&
  person:*:isowner = pc:$0 &&
  person:$1:locatedat != pc:$0:locatedat =>
    person:$1:locatedat -> pc:$0:locatedat
;

pc:*:status ::
  pc:$0:status != BUSY &&
  person:*:isowner = pc:$0 &&
  person:$1:locatedat = pc:$0:locatedat =>
    person:$1:locatedat -< pc:$0:locatedat
;
```

6.1.2 Learning environments: Itechcalli

Within the Itech Calli project we created another laboratory at the Instituto Tecnológico Superior of Zacatecas North, Mexico. This space was designed as a Ubiquitous Computing classroom to be replicated in different places within the state. Thus, the underlying idea of Itech Calli is to create a set of educational spaces or classrooms interconnected in such a way that the students and teachers could be located in any of them, having a remote class.

Being designed as a classroom, the set of capabilities of this environment is different from those of AmIlab. Eight rows of student chairs, each equipped with pressure sensors and a red button, a blackboard and two displays, a 50–inch display beside the blackboard (the virtual window) and a secondary display at the teacher's desk, created a physical environment similar to a classroom (see Figure 6.2). In addition, like AmIlab, it is equipped with lights, switches and cameras.

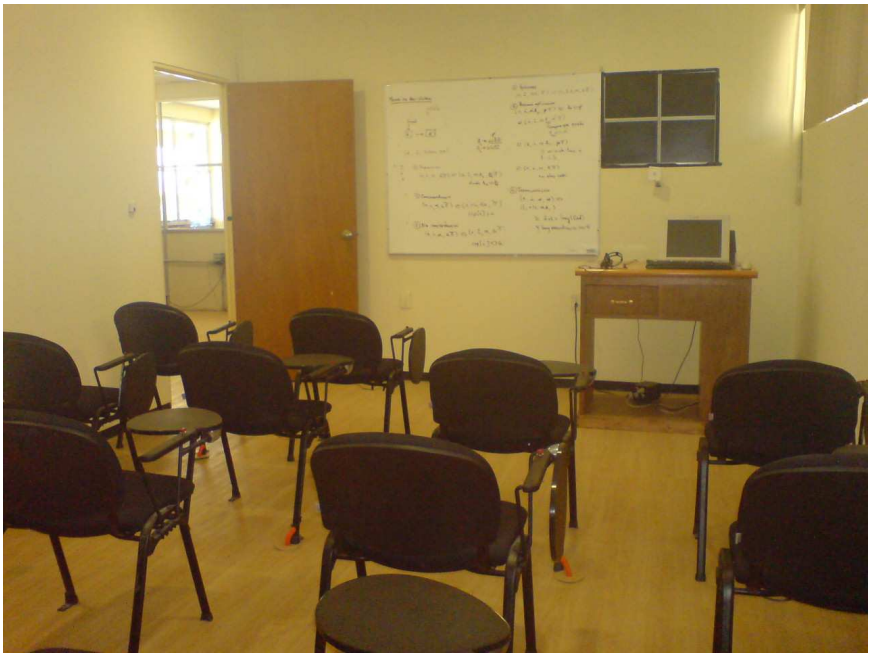


Figure 6.2: A view of the Itech Calli laboratory's living room at the Instituto Tecnológico Superior Zacatecas Norte, Mexico

The set of agents deployed in it has quite a different set of rules, such as those for controlling a slide presentation from different places (even remote places or simultaneous places) or to activate and show some camera image in different displays as events occur (such as a student pressing a button to ask a question). Following the natural structure of a classroom environment (contrary to the freer structure of a living space), the agents were more structured and, normally, associated with a particular activity such as *exam*, *class*, *debate* or *free* which allowed the creation of meta–agents to manage the flow of activities (and their associated preferences) held inside the space. In addition, due to the coordination requirements of a distributed classroom, the use of wildcards was more frequent (see Rule 13).

6.1.3 Security environments: Indra

Finally, through the UAM–Indra Cátedra and as part of the Hesperia project, we deployed a third laboratory at Indra's Madrid facilities. The aim of the project is to develop security technologies for home security and public spaces, and the lab was designed to study security strategies in spaces with sensitive content (books and computers in the first experiments). Here, the natural hierarchies were strict and centralized, and so was the agent structure, which was designed top–down to deal with problems such as sending alarms for unauthorized access to documents

Rule 13 Example of a rule for a ubiquitous classroom in which it is specified that "When the red button of a chair is pressed (meaning that a student has a comment to make) the virtual windows of all the rooms involved in the same ubiquitous event will show the room in which the student pressed the button"

```
chair:*:red_button ::
  chair:$0:red_button = 1 &&
  room:*:contains = chair:$0 &&
  room:*:isinvolvedin = room:$1:isinvolvedin &&
  camera:*:locatedat = room:$1 &&
  vWindow:*:locatedat = room:$2 =>
    vWindow:$4:shows => camera:$3
;
```

or spaces or defining dynamic access rights, consulting times and securing reading places according to context.

This scenario provided a test bench for the language's most complex structures. Due to its secure nature, the programmers' degree of expertise was expected to be higher and thus, we could exploit the potential of both Wildcards and Timers (see Section 4.1.3).

The context–dependent event composition (see Section 4.1.4) was found here in many scenarios. For example, in one scenario there was a one–person zone with a drawer monitored through cameras from a security room. If somebody opened and closed the drawer with no security personnel present at either time to watch what had been taken out or placed inside, a *high risk event* was detected. If at both times a security guard was present then a *low risk event* was detected, and if a security guard was only present at one of the events then a *medium risk event* was generated. Thus, when somebody opens the drawer, the composite event generated will depend on whether or not a security guard is present at the time of closing. Given that the maximum time for leaving the drawer open is 5 minutes (otherwise the alarm is triggered), the code for detecting that context–dependent composite event can be seen in Rule 14.

In addition, for the Hesperia project, we developed a PDA program to show alerts generated on the Blackboard. The *alert* type is a subtype of *message* and, as such, it has the *type*, *priority* and *msg* properties. In addition, it may have the relations *from*, *to* and *locatedat*. The PDA program was subscribed to the creation of *alert* entities so, when a new alert was added to the blackboard, it received a notification with all its attributes (see [60] for a detailed description of the subscription mechanism). This program allowed the user to set some preferences about what kind of alerts should be sent according to the type of alert and priority. In addition, a login screen was used to identify the user so

Rule 14 Example of a rule for detecting a context–dependent composite event in which, after opening a drawer under surveillance, closing it generates a low risk event or a medium risk event according to whether or not it is under surveillance.

```
drawer:security:status ::
drawer:security:status = OPEN &&
person:*:locatedat = room:security_room =>
TIMER 5m 1
   { alarm:security_drawer:status := ON;}
   {
       drawer:security:status ::
       drawer:security:status = CLOSE &&
       person:*:locatedat = room:security_room =>
           <rules to apply to or logging of a LOW risk event>
           &&
           TIMER.kill
       ;

       drawer:security:status ::
       drawer:security:status = CLOSE &&
       person:*:locatedat != room:security_room =>
           <rules to apply to or logging of a MEDIUM risk event>
           &&
           TIMER.kill
       ;
   }
;
```

only alerts directed to the user were shown in the interface (this information corresponds to the *to* relation) (see Figure 6.3).

Alerts, as a reactive behavior, were generated through the use of the ECA–rules described in this work, allowing the creation of different agents, codifying preferences for different persons, domains and scenarios. These agents were activated and deactivated through the use of meta–agents, activating, for example, all security agents when nobody was in the room and deactivating the security agents for particular persons as they entered the room (and consequently do not need to be notified) (see Rule 15).

The rules for creating the messages used the different elements of the environment and some timers, according to preferences. Alerts where created through the *Add entity*, *Add property* and *Add relation* grammar operators (see Section 4.1.3). □

Summarizing, each environment presents a different challenge, from daily living, working, researching, teaching or managing security. In all of them the

Rule 15 Example of the rules of a meta–agent in charge of activating and deactivating the personal security agents of lab B403 members, in charge of sending alarms related to the lab, as they leave and enter the environment, respectively.

```
person:*:locatedat ::
person:$0:locatedat = room:lab_B403 &&
person:$0:belongsto = group:amilab &&
agent:*:is_owner = person:$0 &&
agent:$1:task = security &&
agent:$1:locatedat = room:lab_b403 &&
agent:$1:task = ACTIVE =>
agent:$1:task = INACTIVE
;
person:*:locatedat ::
person:$0:locatedat != room:lab_B403 &&
person:$0:belongsto = group:amilab &&
agent:*:is_owner = person:$0 &&
agent:$1:task = security &&
agent:$1:locatedat = room:lab_b403 &&
agent:$1:task = INACTIVE =>
agent:$1:task = ACTIVE
;
```

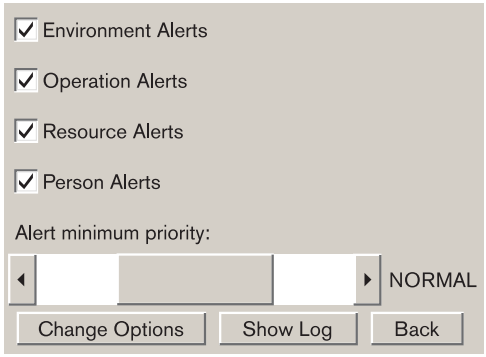
easiness and naturalness was used in a different way: defining preferences and policies, coordinating, prototyping research ideas or obtaining new high–level context information quickly and easily. These challenges served to test the potential of the base–language and its easiness, the complex operators and their descriptive power, as well as the flexibility of our modular architecture to deal with different classes of social organization.

6.2 User Interfaces

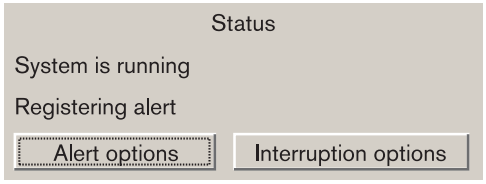
6.2.1 A Graphical User Interface for programmers

In order to facilitate rule creation, we developed a basic GUI. This GUI is not supposed to be an end–user interface, but it eases the process for individuals with low programming skills by freeing the user from having to know the grammar of both the Blackboard and the agent rule mechanism. To do this, the GUI automatically creates a navigation panel. The navigation is by location, so it begins with the largest container in the Blackboard (the University in our case) and allows for location–based navigation by double clicking to go inside a location to see what other spaces are contained in it (e.g. from buildings to floors, from floors to rooms and from rooms to areas). This location hierarchy is automatically

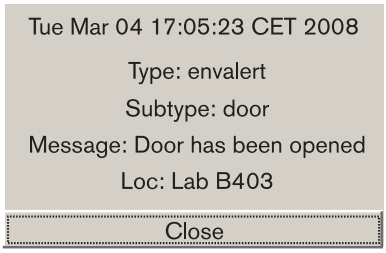
(a) The login screen is used to identify the user to send those alerts sent to him/her



(b) The program allows configuring the type of alerts to receive, and the minimum priority level to receive them



(c) The program in the PDA allows configuring what type of alerts have to be shown and when



(d) Alert messages show the alert type and subtype, time and place and the message attached to the alert

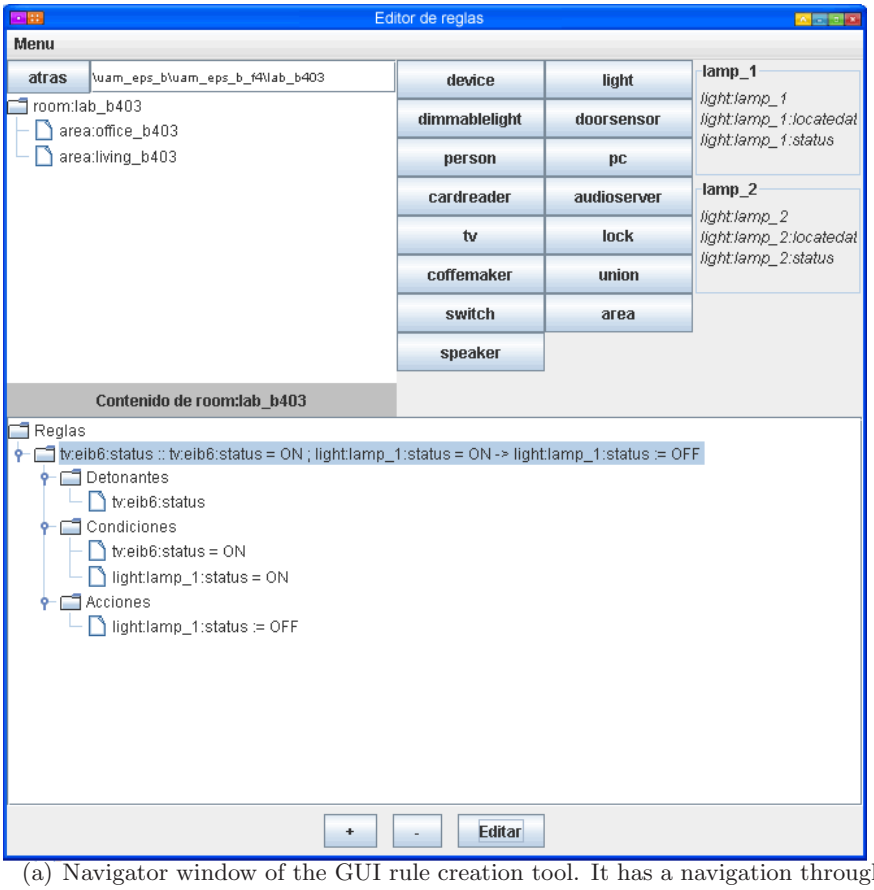
Figure 6.3: Screen captures of the alert PDA software. 6.3(a) shows the startup login screen 6.3(c) shows the main window of the program, 6.3(d) shows an alert received and 6.3(b) shows the configuration screen. This program was coded by Carlos Pimentel.

extracted from the Blackboard through the relations “contains” and “located at”.

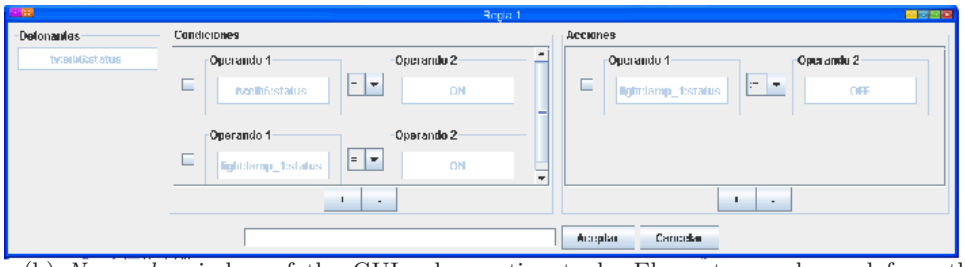
A list of the types is presented in the upper middle part of the interface according to the objects within the selected location (see Figure 6.4(a)). If the user clicks on a type, a list of all the elements of that type, its properties and relations, appears in the upper right part of the interface.

The bottom part of the interface is for rule management and contains a list with all the rules that have been created and buttons to add, delete or modify rules. When adding or modifying a rule, a new window opens with three parts (triggers, conditions and actions), as shown in Figure 6.4(b). Operators and comparators for the actions and conditions are chosen from a list, while the values can be established by dragging and dropping from the Navigation window.

While this interface is not designed for end–users, it has been tested with first–year students with no prior programming knowledge, allowing them to write their own programs after a brief introductory lesson.



(a) Navigator window of the GUI rule creation tool. It has a navigation through location panel on the top left part of the window, the list of the types of elements present in the chosen location in the top middle, the list of the elements of the selected type present in the location in the top right and the list of created rules at the bottom



(b) *New rule* window of the GUI rule creation tool. Elements are dragged from the navigator window and dropped in the corresponding area of this one (triggers, conditions or actions)

Figure 6.4: A snapshot of the windows of the GUI for rule creation. This program was coded by Carlos Pimentel.

6.2.2 The Magnet Poetry metaphor

In addition to this GUI we developed an initial prototype for an end–user interface. In order to be suitable for end–users, the interface must be easier than the one presented, and the means for expression more natural. Thus, using terms such as *light:lamp_1:status* is not acceptable.

When deciding what kind of interface metaphor to use, we considered different types of natural interactions, such as natural language, icon–based and tangible user interfaces (TUI). The natural language interaction was the preferred choice from the beginning, since it did not require any kind of extra abstraction from the user (contrary to an icon–based interface). Nevertheless, precisely because natural language is ordinarily used for everything in users' daily lives, and given that not everything users perceive as being part of the environment is part of the intelligent environment or can be done in it (e.g. moving objects, opening the door or telling where a particular pair of socks is), a non–constrained interface such as natural language may lead to uneasy situations in which the end–user does not know what can be said and what can not, not being able to establish a reasonable cause for it.

On the other hand, we believe that tangible interfaces, when correctly designed, present a fairly natural means of interaction. Nevertheless, TUIs are prone to number and expression problems. That is, when the set of concepts to manage grows in number, TUIs become fairly complicated to use: they either have to account for a huge number of physical pieces (difficult to find) or they have to use an abstract mechanism to represent more than one concept with only one piece (difficult to understand).

Thus, we opted for a GUI, based on a TUI using natural language interaction: a "fridge magnet poetry" based GUI.

The magnetic poetry metaphor is based on small magnets, each of which has something written on it. Poems (sentences) can be constructed by arranging the magnets on a metallic surface. One of the strengths of the metaphor is that, since the words are already written on the magnets, the expression is more accurate and the capabilities of the environment can be seen at glance.

This concept of a magnet poetry based GUI was already applied by Truong et al in CAMP [116]. It gave users complete freedom to use the tokens to build sentences. In order to ease the "writing" process, CAMP provides some search services as well as a "translator" to give feedback to the user on how the system is understanding the sentence.

We, like Truong et al. [116], are very interested in domestic environments and firmly believe that the magnetic poetry metaphor might succeed in some interesting scenarios of this particular domain: it is appealing to non-programmers and, more importantly, it allows end-users to program from the beginning in an efficient and effective manner. Hence, with such a low learning curve, barriers are

overcome before the user decides to discard the metaphor. In addition, the naturalness of the "when" "if" "then" structure of the rules, as well as the artlessness of the "entities" "properties" and "relations" (nature of the rules' bricks), makes the language human-readable from the very beginning.

Figure 6.5: Base layout of the Magnet Poetry GUI, showing the different zones for different types of magnets (nouns, verbs, values and links) and the working zone (bottom of the figure) with two parts, a rule zone and a working space around it. This program was partially coded by Amanda Vidal.

Currently, we have not devised any other environments where the metaphor might be applied. Domestic environments are suitable for occasional programming. However, production-oriented environments, such as an office, a laboratory or a store, require more expressive tools.

The main difference between CAMP and our approach lies in how the process of building a rule statement is implemented. In the case of CAMP, users build their statement without imposing any restriction on the selection and arrangement of the tokens. During the rule composition, tokens can be chosen freely and they can be placed in any part of the statement. Once a phrase is completed, several parsing techniques are applied to transform the original statement into a computer-readable one. This includes rewording using dictionaries, decomposing the phrase into a collection of sub-clauses, or removing redundant information. Additionally, CAMP includes a status bar where the user is given feedback with the interpreted statement. In this manner, the user can check incorrect or missing statements, and, if required, fix them by changing certain tokens. What is more, ambiguity and missing parameters in the rephrased description are flagged or set

to predefined values. Dimensions flagged as missing can be indicated to the user providing her with the opportunity to refine the description. Missing information in the final description is replaced with default predefined values.

On the contrary, we argue that it is necessary to steer the user during the statement building. This guidance has to include both token selection and placement. Regarding the former, before each new step in the statement composition, the set of tokens is filtered according to previously placed tokens. Thus, only tokens that can be placed at the end of the ongoing sentence are shown, removing those that, if chosen, might lead to a wrong sentence. This results in a smaller set space, easing the search process and giving a better formed sentence (the user can only select from the correct set of tokens). In addition, tokens are displayed in different cabinets, corresponding to *verbs*, *nouns* and *values* to decrease the search space through positioning (see Figure 6.5).

Therefore, our methodology allows the user to construct less natural statements, although the expressiveness remains the same. For instance, Table 6.1 shows how some of the example statements posed by Truong et al. might be reformulated in our proposal. It is interesting to note that Truong's and Dey's studies reveal that people find it natural to interact with technologically–enriched homes as commanded systems. Thus, command–like sentences are natural to users when instructing their homes. While Truong et al. believe that users want to express themselves freely, we believe that they want to express themselves easily and accurately. Although "freely" normally implies "easily", we prefer to explore other means of "easy expression" that are more accurate than "free expression", believing that the feeling of freedom experienced by a user is hardly affected if the language is powerful and easy enough. In addition, the satisfaction experienced by being able to express oneself freely can be overcome by the frustration of a system unable to correctly interpret the sentences.

Besides a steered composition of sentences, our system also presents a feedback process through which users can see the sentence in the machine–like language by clicking on the "translate" button of the interface. While the machine–like language is not as easy to read as the one that uses magnets, users do not have to write on it but just use it to check the correctness of the sentence, if desired. Knowing what should be written makes the understanding process that much easier.

According to the above, ambiguous sentences are less probable, since we provide several mechanisms for verifying that composed rules are semantically unequivocal. This implies that the task will be correctly accomplished on the first attempt, avoiding disturbing iterations. Moreover, we surmise that as a CAMP user gains more control over the interface, he will adapt his expression to the

Table 6.1: Several statements represented as CAMP approach and our approach.

CAMP statement	**Our statement**
"record video everywhere Saturday night"	**When** Saturday night **Then** record_video_of everywhere.
"record picture in Billy's bedroom at night"	**While** Billy is_in bedroom **if** at_night **Then** record_picture_of Billy
"record 1 picture every 4 minutes in Billy's bedroom every night until morning stop"	**While** Billy is_in bedroom **if** at_night **Then** record_picture_of Billy and time_between_pictures equals 4 minutes
"always show me where baby Billy is"	show me Billy's_location
"always record picture of baby Billy and display at my location a picture of baby Billy"	Record_picture_of Billy and show me picture_of Billy

internal representation of the tool rather than continue to use his own words. Essentially, users learn how the system builds statements thanks to the feedback they receive whenever a new rule is inserted. In other words, the user learns how to tune the system to obtain a better response. As such, we believe that users will prefer to become familiar with the system's expressions rather than let the system interpret (or misinterpret) their words. This can be seen in the way people perform searches in free–expression searchers such as Google^{TM}.

This interface is based on a "Magnet poetry" grammar and a parser for translating it to the ECA–rule language grammar. A simple language was created to test the concept within a limited domain. This language was created to allow users to configure some switchable and adjustable objects (such as the light, the speakers, the coffee maker, the volume of the speakers or the radio) according to their state, the state of the door and some location information. The EBNF grammar used for this example is shown in Rule 16.

This EBNF grammar is used to generate a "Step by Step" analyzer to guide the composition of sentences. Thus, when a token is input, it returns the list of possible subsequent tokens allowed by the grammar. Once the user is finished composing the sentence, the resulting tokenized sentence is passed to a translator for conversion to base–language rules. These rules can then be loaded into an agent. This process is illustrated in Figure 6.6.

6.3 Integration with other technologies

One of the main goals of the AmIlab laboratory since its inception has been to integrate different technologies smoothly and easily. This integration has focused on hardware technologies. EIB, Phidgets, RFID, X10 and others were integrated

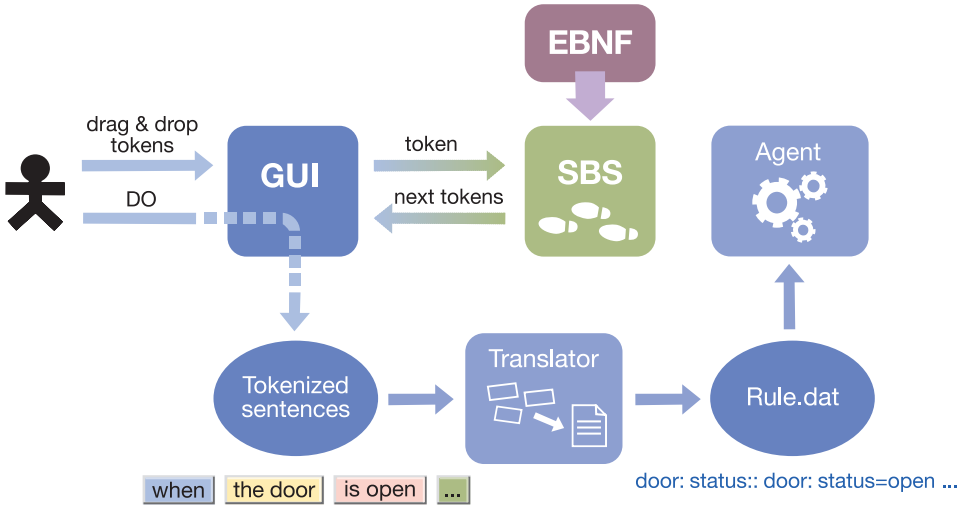


Figure 6.6: Execution flow for translating sentences generated in the Magnet Poetry GUI to the ECA–rule kernel language. First an EBNF grammar is used to generate a Step by Step analyzer to guide the composition of sentences. The tokenized sentences produced by the end user are translated to the ECA–rule language and loaded into an agent.

through the blackboard. Some of them, such as the Phidgets, were intended to extend the end–users' creation capabilities of the agent system to the hardware domain. This hardware integration was done through the transparent view provided by the Blackboard, but the agent system allowed us to easily merge new software technologies and systems, hence combining their potential with the already existing context–aware capabilities. Thus, we incorporated the anthropomorphic virtual character "Maxine" [39] of the University of Zaragoza, Spain, to deliver messages in the environment from a wall–mounted 50" display. What to say, when to say it and the mood to use when saying it were behaviors programmed through the agent mechanism. All the low–level complex behaviors such as lip synchronization or facial expressions were part of the state–of–the–art work incorporated by the Zaragoza team (see Section 6.3.1). The last example of synergy can be seen in our work with the Embedded Systems Laboratory of Lancaster University (UK), in which we combined their Smart–its technology and steerable projection system [84] with the blackboard representation and agents structure to create a guided cooking scenario in which the system guided the user to cook the selected dish by projecting over the different elements involved in the recipe (e.g. salt, pans or stove) (see Section 6.3.3). All the system logic was again delivered by the agents' mechanism while the new sensing and projection capabilities were the fruit of the work done at Lancaster University.

6.3.1 Integration with anthropomorphic figures

Focusing on multi–modality and social interaction in Ambient Intelligence, Maxine [39] is a script-directed engine, of the Universidad de Zaragoza, for the management and visualization of 3D virtual worlds, written in C++ and based on a set of open source libraries. Those virtual worlds may include actors with synthesized speech, synchronized lip movements and different moods.

All the elements that constitute a virtual scenario are defined and managed through a scene graph including images and texts (bmp, gif, jpeg, pic, png, rgb, tga, tiff and alpha channel), simple geometric primitives and complex geometric models (3DS, flt, lwo, md2, obj, osg), simple lights, 3D and ambient sound, animated characters (Calc3D), animated actors (provided with voice synthesis and facial animation with lip–sync) and synthetic voices (with voice selection, volume control, speech speed, insertion of pauses, tone, word emphasis, pronunciation specification). The engine also manages auxiliary elements like cameras.

While this system facilitates the creation of applications with 3D virtual worlds (such as a PowerPoint presentation [39]), its focus is on programmers. Thus, we decided to try to integrate it into the environment as another part of the world, to allow users to take advantage of a virtual character for their own purposes.

To accomplish this, we created a driver and modeled Maxine as a virtual person in the environment with three properties: *say*, *mood* and *emphasis*. The *say* property was inherited from the previously existing *synthesizer* type, while the other two were added to exploit the visual capabilities of an anthropomorphic character. The *mood* property could take one of the following values: “anger”, “disgust”, “fear”, “joy”, “sadness”, “surprise” and “neutral”, while the *emphasis* ranged between 0 and 1.0. Then, Maxine was shown on a 50–inch wall display in the living room.

First of all, Maxine was integrated with the oral direct interaction mechanism [86] so users had a visual reference to command the environment, whereas before they had to speak to the void, which lead to uneasy situations when the speech recognition failed.

A Study of the Use of a Virtual Agent in an Ambient Intelligence Environment [89] reported that almost half of the subjects (48%) preferred to interact with a virtual agent in their conversation with the ambient intelligence environment, only 14% of them considered that they preferred to carry out the interaction without the support of the virtual agent and 38% did not express any specific preference. In addition, one of the most important consequences of driving the interaction through an avatar was related to trust. Thus, when the speech recognition failed, the avatar provided a visual so people naturally placed the responsibility on it, leaving their trust on the rest of the system unaffected. This fact is strongly related to the “spreading responsibilities” issue of Section 3.3.3

that led to the modularization of agents in the first place.

The integration with the oral direct control mechanism turned Maxine into an input device. Modeling it in the Blackboard also turned it into an output device that could be harnessed by users in their context–aware applications.

Some examples, freely developed by first–year graduate students with no prior programming knowledge, were aimed at greeting people as they entered, notifying of events, making comments about weather conditions or changing the environment with Maxine's mood. While not everybody will want to have such behaviors in their personal homes (specially those in which a bad–tempered avatar can turn the lights off unexpectedly), it gave an idea of the flexibility and creation capabilities of the rule–based system and its integration. An example of some of these rules can be found in Rule 17.

This experience reassured our confidence in the Blackboard architecture for how easy it was to integrate new software into it, and in the Agents-Rules mechanism for how simple it was for the first–year graduate students to translate a wide range of ideas into operating behaviors, even though they had no prior programming knowledge.

6.3.2 Integration with Phidgets

Probably the most important feature of Intelligent Environments is their hardware-software duality. The computing/interaction power leaves the computer environment to merge with the natural human environment. Doors, sofas, coffee makers, telephones, tables and more become a part of the macro–computing entity that is the environment, serving as inputs, outputs or processing units.

In the Department of Computer Science at the University of Calgary, Chester Fitchett and Saul Greenberg developed a system to deal with the problem of easily incorporating physical devices into user interfaces: the Phidgets. Analogous to physical widgets, the Phidgets —or physical widgets— "abstract and package input and output devices, hiding implementation and construction details while exposing functionality through a well defined API" [52]

As with Maxine, while Phidgets facilitate the creation of physical interfaces, its focus was on programmers. In our case, our work on automatic adaptation handles the problem of programming a Smart Environment by a non-programmer and our Blackboard middleware deals with the problem of integrating new devices into the environment.

After developing a first prototype of the multi–agent mechanism for indirect control, we incorporated two first–year undergraduate students to AmIlab to test the ease with which users with no prior programming knowledge could create new sets of behaviors. To do this, we introduced them to the AmIlab living room and briefly explained (without mentioning the acting and perceiving capabilities of the environment) what kind of objects there were (i.e. a TV, a telephone, a

radio, a coffee maker, two sofas, a CD player...). After that, we told them that the room was intelligent and could act on and perceive the room. In addition, they were told that the environment could be commanded, which required them to write the set of behaviors they wanted the room to have (as if it were their own room).

Even though the experimental laboratory had different types of perceiving and actuating devices, we found that their number and location was not wide enough to support their initial first ideas. In other words, thinking about them as end–users in an intelligent room, they not only wanted to create new behaviors but to sense and actuate it in places were the infrastructure did not allow them to do so.

To let them do so, we wanted to create a mechanism for rapidly and easily extending the architecture of the environment so that end–users could adapt the hardware in their homes, as they do with software, through the agents mechanism, according to their needs.

The Phidgets system provides a board, called InterfaceKit, with eight slots, to which different kinds of sensors can be connected. We used the InterfaceKit as the base for our augmenting hardware experiment and created a set of drivers for different types of Phidgets sensors, with each analog sensor having different drivers with different accuracy ranges (binary, 3 values, 5 values...). Since the sensors were physically tagged with their type (e.g. Pressure, Light, Temperature or Distance), users could easily identify them and tell what accuracy range they wanted. The second step was to associate the sensor with a real object that was to be augmented. This was done through a configuration file in which, for every slot of the InterfaceKit, there was a four–slot template. The slots corresponded to the sensor being used (type of sensor and degree of accuracy desired), and the type, entity and property they wanted to associate with the sensor. Thus, for sensing whether a sofa was occupied or not, they plugged a weight sensor on a slot of the InterfaceKit (slot number 0 in this case), placed the sensor under the sofa cushion and created a configuration file with values *binary_weight*, *furniture*, *sofa_1* and *occupied* in the 0 slot template (see Table 6.2).

Table 6.2: Example of Phidget configuration file for adding a weight sensor to a sofa to sense if it is occupied or not

Sensor:	binary_weight
Driver:	binary
BBtype:	furniture
BBentity:	sofa_1
BBproperty:	occupied

Then, we developed a program that, given a configuration file, associated the sensed value of the sensors to the Blackboard properties of an entity, creating it automatically if it did not exist in the Blackboard.

In addition to other works such as [63], this mechanism supplies, besides the sensors' drivers (with their hysteresis processes, thresholds and other low level procedures), a direct abstraction from low–level context (i.e. a sensor is pressed) to high–level context (i.e. the sofa is occupied) that is more suitable to work with.

The combination of hardware and software adaptation allowed the students to create extremely interesting applications in under fifty hours. Next, we provide a representative set of examples in a form similar to the *Recipe* suggested by Mark W. Newman et al [12]: with some "ingredients" —those elements added to the environment— and some "procedures" —the set of behaviors encoded in the agents.

Augmented phone

Purpose: to display relevant information on the phone and to react to its status —on–hook/talking.

Ingredients: An LCD Phidget display and a 5–mm presence Phidget sensor.

These elements were added to the Blackboard context as two new entities: a display called *telephone display* and a device called *telephone* with an *in_use* property. The configuration file is shown in Table 6.3. We should note that the Phidget LCD comes with an integrated InterfaceKit, so there is no need to add another. The LCD needs no configuration file and is created automatically when running the driver. Users only have to give it a name.

Table 6.3: Code of the configuration file for the augmented telephone. *Sensor* indicates the type of sensor plugged into the first slot of the interface kit. *BBtype*, *BBentity* and *BBproperty* indicate, respectively, the ontology-type and names of the entity and property to which the sensor is assigned.

Sensor:	binary_5mm
BBtype:	device
BBentity:	telephone
BBproperty:	in_use

Alternatively, a *Telephone Agent* was created to control the telephone with the following behaviors:

- When the telephone is picked up, if the TV or the radio are on then lower their volumes.
- When there is a change in Manuel's location, then show the new location on the telephone display.
- When the telephone is picked up, then deactivate the doorbell.
- When the telephone is on–hook, then reactivate the doorbell.
- When somebody rings the doorbell, if the telephone is picked up then show an alert on the display.

Is it possible to see that the new ingredients are seamlessly integrated with the already existing elements of the Blackboard.

As a measure of complexity, for developing the phone, the students had to write nine lines of "code": four in the configuration file and five corresponding to the rules, using the GUI.

Augmented sofa

Purpose: use the sofa as a control center in which it is possible to receive and generate general information and, additionally, perceive when the sofa is occupied to activate the control center.

Ingredients: Light and temperature sensors (for lab_b403's temperature and light properties), an LCD display (for the arm of the sofa: sofa_display) and a pressure binary sensor (for the occupied property of the sofa).

Those elements were controlled by two different agents, one in charge of displaying weather information when nobody is seated on the sofa and another for using the display and buttons as an interface to explicitly notify of an activity in progress. The latter will be presented in section 6.3.2. The former is composed by the following rules:

- When the sofa is unoccupied then show the temperature and light values on the sofa_display.
- When the temperature value changes, if the sofa is unoccupied then update the temperature value on the sofa_display.
- When the lighting value changes, if the sofa is unoccupied then update the lighting value on the sofa_display.

Creating scenarios

In the previous section we showed two examples of personalization that focused mainly on adding hardware but, as already stated, the power of the system comes precisely from the synergistic combination of both personalizing hardware and software.

One of the first things the students wanted to do was to adapt the environment to some activities, creating sets of behaviors to apply according to previously defined scenarios. These scenarios were mainly linked to specific activities such as relaxing, working, watching TV or taking a nap.

The problem involved the difficulty of automatically extracting context information of such an extremely high degree of abstraction —an example of this difficulty can be found in Nuria Oliver's work [99]. To this end we suggested that they implement interfaces distributed along the room (smoothly merged into it) for users to explicitly specify the activity they were involved in.

The scenarios were created as different agents, one for each (i.e. relaxing, watching TV, napping, or working) and a meta–agent in charge of activating and deactivating them according to the activity in progress (see Section 5.4.3). Thus, for every agent the meta-agent had two rules of the type:

- When the activity changes to relax, if the relax agent is deactivated then activate the relax agent.
- When the activity changes, if the relax agent is activated and the activity is not relax then deactivate the relax agent.

As an example to illustrate the types of behaviors encoded by the students in the agents, we show some of the rules of the relax agent:

- When the relax agent activates, if the main light is turned on then turn on the secondary light and turn off the main light.
- When the relax agent activates, then deactivate the doorbell agent.
- When somebody rings the doorbell, then show an alert on the sofa_display.

These rules give an idea of three different types of behaviors: on–load rules, meta–agent rules and normal rules referencing the new "phidgeted" elements.

The interface for specifying the activity in progress was developed over the augmented sofa, re–augmenting it through the following recipe:

Purpose: Create an interface to specify the activity in progress.

Ingredients: Two touch sensors for interacting (corresponding to entities *sofa_ button_change* and *sofa_ button_select*)

Procedures:

- When the sofa becomes occupied then show RELAX on the display.
- When the sofa_ button_change is pressed, if the sofa_display shows RELAX then update it to WORK.
- When the sofa_ button_change is pressed, if the sofa_display shows WORK then update it to SIESTA.
- When the sofa_ button_change is pressed, if the sofa_display shows SIESTA then update it to WATCHING TV.
- When the sofa_ button_change is pressed, if the sofa_display shows WATCHING TV then update it to RELAX.
- When the sofa_ button_select is pressed, then set the room activity to the display value.

It is possible to see that the first five rules define a circular menu, while the last one is in charge of the activity selection.

It should be noted that the agent was built incrementally as new necessities arose, from a one button and no display interface, to just choosing the WATCHING TV activity, to the circular menu with the two buttons and LCD display presented above.

Since the implementation was gradual and alternated with other activities, it is hard to estimate the total amount of time spent in developing the interface. About an hour is probably a good approximation.

Our experience with students helped us to evaluate how creation tools empower creativity, as well as how easy it is for non–programmers to create complex devices when they are provided with easy tools.

Additionally, as in the Maxine experience (run at the same time), this experience reassured our confidence in the Blackboard architecture for how easy it was to integrate new hardware into it, and in the Agents-Rules mechanism for how simple it was for the first–year students to translate a wide range of ideas into operating behaviors, even though they had no prior programming knowledge. Finally, it convinced us of the necessity of physical building blocks, easy to use by the end-user, to augment or create ad-hoc devices in contrast to —or in addition to— market physical devices.

6.3.3 Integration with steerable projection

While the Phidget and Maxine experiences served to test how to integrate hardware and software technologies with the Blackboard–Agents platform, the following experiment shows an integration with an already existing state–of–the–art hardware–software technology.

The University of Lancaster developed a system to dynamically project images on objects, looking for "cooperation between mobile smart objects and projector-camera systems to enable augmentation of the surface of objects with interactive projected displays" [84]. The system was based on the Smart–its [49], small devices integrating different sensing, communication and processing capabilities and combined with a steerable projector–camera system to track objects visually as they move around, and project interactive interfaces and information on the object surfaces taking into consideration the shape and orientation of the object so the image is shown undistorted to the user.

In this sense, the objects were self–describing, carrying all their information in an Object Model. The projector–camera system provided a projection service that any object could request by sending its Object Model and the image to be displayed. While the Object Model carried information exclusively about the object (i.e. a unique object identifier, appearance knowledge, 3D model and sensor knowledge), the location and orientation of the object had to be supplied by the projection–camera system (see Figure 6.7).

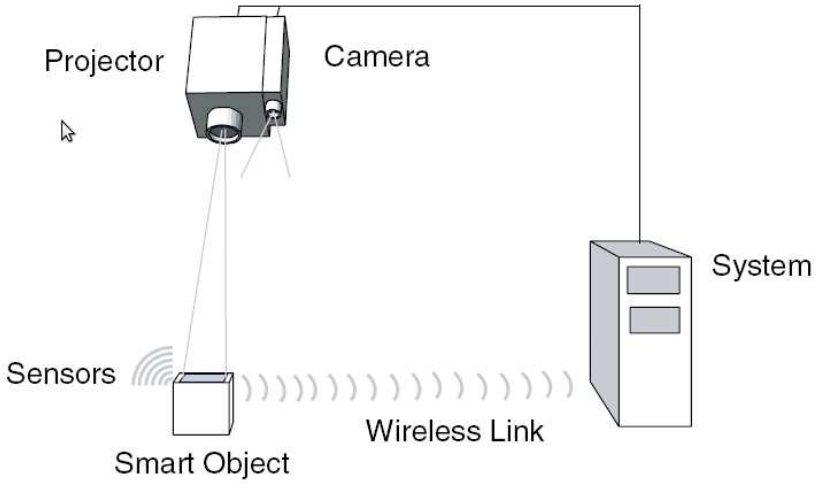


Figure 6.7: Cooperative Augmentation of Smart Objects with Projector-Camera Systems. Extracted from [84]

External context information, on the other hand, was harder to retrieve and the programming mechanism difficult and slow. In order to expand the system so context–aware applications could be easily deployed and benefit from the projection capabilities too, we integrated those systems into the Blackboard–Agent platform.

The experiment was conducted as a cooking scenario, in which the system guided the user through the cooking process by displaying visual messages on the objects. Thus, for example, when water was boiling in a pot, the system

would project a message on the pot ("Warning: hot water. Add egg now") and a message on the egg box ("Add one egg to the pot") for the "boiled egg" recipe.

To do so, we added to the Blackboard new entities corresponding to the new objects in the kitchen (i.e. *stove*, *medium_pot* and *eggs*), with their corresponding properties and a parameter with their unique id's. In addition we developed a driver that, upon the creation in the Blackboard of a message entity of the type *REQUEST PROJECTION*, sent a request to the projector using the same format that a smart–it would have used. As an example, a message with the property *type* set to *REQUEST PROJECTION*, and a relation *locatedat* linking it to the entity *medium_pot* was used by our driver to send a projection request to the projector, as if it were from the Smart–it in the medium pot. The image to project was codified as a URL in the *text* property of the message.

With the elements modeled in the Blackboard and the driver running, the recipes were programmed as rule–based agents. Having one agent per recipe, multiple recipes could have been handled through meta–agents. An example of a rule for displaying a message in the pot can be seen in Rule 18.

It is important to note that all merging and programming (including the driver) was done in less than a full work day, while most of the time was spent in associated tasks such as creating the hardware (pans, stove, salt...). In addition, while the Smart–its, camera–projector system and recipe agents were running in Lancaster University (UK), the Blackboard on which they were modeled (and to which the agents and drivers were subscribed) was in the Universidad Autónoma de Madrid (Spain) (in the first pan-European boiling experience).

In conclusion, this experiment proved the potential of the Blackboard–Agents mechanism as a mechanism for integrating/coordinating/merging already existing hardware and software systems.

All of these experiments allowed us to test our hypothesis that this system will subsume the application–dependent and flexibility–simplicity inverse relation problems. The most important conclusions derived form them will be discussed in the following section.

Rule 16 Grammar for a simple example to control some elements of the environment with the Magnet Poetry GUI.

```
SENTENCES           : (SENTENCE) +;
SENTENCE            : 'WHEN' CONDITIONLIST CONDITIONS? 'THEN'
                       ACTIONLIST;
CONDITIONS          : 'IF' CONDITIONLIST;
CONDITIONLIST       : CONDITION ('AND' CONDITIONLIST)*;
ACTIONLIST          : ACTION ('AND' ACTIONLIST)*;

LOCATION            : 'HOUSE' 'KITCHEN' | 'LIVING ROOM' | 'SOFA';
PERSON                : 'PABLO' | 'MANUEL' | 'GERMAN' | 'XAVIER';
OBJECT              : OPENCLOSEOBJECT | ADJUSTABLEOBJECT
                      | SWITCHABLEOBJECT;

CONDITION           : CSWIT | CADJU | COPEN | CCONT;

CSWIT               : SWITCHABLEOBJECT (SWITCHABLESTATUS
                      | LOCATIONOP LOCATION);
SWITCHABLEOBJECT    : 'THE_LIGHT' | 'THE_KITCHEN_SPEAKER'
                      | 'THE_RADIO' | 'THE_LIVING_SPEAKER'
                      | 'THE_COFFEMAKER' | 'THE_TV';
SWITCHABLESTATUS    : 'IS_ON' | 'IS_OFF';

CADJU               : ADJUSTABLEOBJECT (ADJUSTABLESTATUS
                      | LOCATIONOP LOCATION);
ADJUSTABLEOBJECT    : 'THE_DIMMER_LIGHT' | 'THE_VOLUME';
ADJUSTABLEOP        : 'IS' | 'IS_NOT';
ADJUSTABLESTATUS    : 'HIGH' | 'LOW' | 'OFF';

COPEN               : OPENCLOSEOBJECT OPENCLOSESTATUS;
OPENCLOSEOBJECT     : 'THE_DOOR';
OPENCLOSESTATUS     : 'IS_OPEN' | 'IS_CLOSED';

LOCATIONOP          : 'IS_IN_THE';

CONTENTOP           : 'CONTAINS';
CONTENT             : PERSON | OBJECT | LOCATION;
CCONT               : LOCATION CONTENTOP CONTENT;

ACTION              : ADJUSTABLEOBJECT ADJUSTABLEACTION
                      | SWITCHABLEOBJECT SWITCHABLEACTION;

SWITCHABLEACTION    : 'TURN_ON' | 'TURN_OFF';
ADJUSTABLEACTION    : 'TURN_ON' | 'SET_LOW' | 'SET_HIGH';
```

Rule 17 Example of rules for using Maxine's capabilities to greet people on entering the room, notifying events, expressing "feelings" on weather conditions and changing the environment based on mood

```
person:mherranz:locatedat ::
  person:mherranz:locatedat = room:lab_b403 =>
    person:maxine:say := "Hello Manu"
;

device:coffeemaker:status ::
  device:coffeemaker:status = OFF =>
    person:maxine:say := "The coffee is ready"
;

room:lab_b403:temperature ::
  room:lab_b403:temperature > 25 =>
    person:maxine:say := "So hot in here" &&
    person:maxine:emotion := DISGUST
;

person:maxine:emotion ::
  person:maxine:emotion = DISGUST =>
    light:lamp_1:status := 0
;
```

Rule 18 Example of a rule for projecting a "Warning: hot water" message on a pot using the steerable projector driver. The CE operator creates a new entity, AP adds a property to the entity and the character # is replaced by a unique identifier (see Chapter 4)

```
device:medium_pot:temperature ::
  device:medium_pot:temperature > 70 =>
    msg CE hot_water#  &&
    msg:hot_water# AP type  &&
    msg:hot_water#:type := REQUEST_PROYECTION  &&
    msg:hot_water# AP text  &&
    msg:hot_water#:text := "file://D:/cooking/hotwater.png"  &&
    msg:hot_water#:locatedat -> device:medium_pot
;
```

Chapter 7

Conclusions

This work has presented a kernel language and structure for indirect control in Intelligent Environments. Both the language and the structure were designed to match the natural human mental processes for expressing and organizing preferences, pursuing a control mechanism through which all the elements of the environment may be used and programmed seamlessly and in a natural way: **Bringing the means for programming the environments closer to how users think about their environments**.

This objective presented two problems in current environments: **unshared information/capabilities** and **unshared interaction mechanisms**. That is, even though different elements of the environment can perceive or act on the environment (e.g. a thermostat or a doorbell), they do not share their information or capabilities with the rest of the environment's elements. In addition, while some of those elements can be controlled or programmed, this is done in different ways. These two problems together lead to a frustrating control experience, increased as the number of perceptive, active, controllable or programmable elements grows in the environment.

In addition, when designing a system to solve these problems, a number of issues have to be considered. These issues, analyzed in Chapter 3 can be summarized by two terms: the **heterogeneity** of users (i.e. heterogeneity of preferences, environments and expertise) and their **multiplicity** (i.e. the coexistence of multiple users, preferences and domains of control).

The underlying philosophy is that the intelligence of the environment must be used to leverage the control experienced by its inhabitants, contrary to other trends pursuing an artificial intelligence, able to autonomously manage the environment on behalf of the user (see Chapter 2). Therefore, the work is based on an **automatic control paradigm**, through which users are able to program behaviors that will, hereafter, execute automatically. Contrary to this paradigm is the

autonomous control paradigm in which the system decides what behaviors are best suited to satisfy the user.

Finally, the work pursues a *UI–independent programming paradigm.* That is, instead of focusing on how to design an end–user programming interface, this work looks to devise a paradigm that captures the end–users' programming essence in a mechanism that can be easily translated to different UIs. Therefore, while different UIs can be designed to meet specific interaction requirements, the programming means will not be altered from UI to UI. In addition, by having such a mechanism, UI–designers can focus just on interface issues.

The programming mechanism is based on two key components: an Event Condition Action (ECA) rule expression language, and a multi–agent modularization structure, respectively designed to deal with the heterogeneity and multiplicity issues.

Regarding the language, an **ECA–rule based language** has proven to be a **natural means of expression**, as suggested by Myers and others (see Section 3.3), especially when programming reactive behaviors by associating an action to a specific context.

Besides, by being human–understandable expressions, they provide a valid base for **explanation** that, in addition to favoring trust (as many authors suggest, see Section 2), has proven to be an essential **debugging mechanism** to improve competence. In addition, ECA–rules provide a feasible ground for machine **learning** (see Section 5.3.2).

Over and beyond the above, having a simplified base–language with a set of extensions to express more complex concepts (such as generality or time), provides a mechanism to isolate complexity, thus **tackling the inverse relationship between flexibility–simplicity** common to most end–user oriented programming systems. In this way, users only have to deal with that complexity that is strictly required by their problems. Moreover, building the complex extensions using the base–language structures and concepts provides a **gentle learning curve** in which every learning step forward eases further improvements.

Finally, by working with Timers (see Section 4.1.3) we realized the potential of using an **event logic instead of an event algebra** to build composite events. Even though this is not strictly related with an end–user directed approach, it presented some interesting questions about the intrinsics of event composition. In this sense, Timers allow a procedural definition of composite events, compared to the more declarative composition of event algebras (in which a number of combinable composite events are declared, e.g. sequence, disjunction or closure) (see Section 4.1.4). In addition, composite events and consumption policies are coded simultaneously in the Timer, allowing for more flexible combinations of different events and consumption policies. The structures for coding the most common event algebras and consumption policies of the literature have been de-

scribed. The questions about whether or not it is appropriate to lose the ease of construction of a more declarative approach and of how far a procedural mechanism can go compared to the former is yet to be studied. Nevertheless, some initial steps have been taken with the definition of two uncommon event compositions: context–dependent event composition and mixed consumption policies (see Sections 4.1.4 and 4.1.4).

Regarding the **multi–agent modularization structure**, this has been successfully used to allow an **organic evolution** of the system's set of preferences, as well as to **spread responsibilities** among different modules (to the benefit of a more trustable and robust system) (see Section 5.1). Furthermore, representing the agents in the Blackboard, and providing them with an "are you alive?" protocol, opened the door to self–recovery strategies to address the breakdown of agents.

In addition, representing the agents in the Blackboard as another part of the system allows for the possibility of creating rules for the activation and deactivation of agents (i.e. of packets of preferences) according to context. That is, to **build hierarchies**. Tagging agents in the Blackboard with contextual information such as who they work for, where they work, what elements they affect or what specific task they are intended to solve, allows users to structure their hierarchies according to their natural structures. In addition, a rule–based language also supports a natural description mechanism to express the prioritization rules of a hierarchy. Not imposing any degree of *interdependence* and providing different degrees of *scale* guarantees that the coordination structures built with this system can have **many possible degrees of complexity** (as natural social structures have).

Finally, combining the coordination possibilities of the multi–agent mechanism with those of the Blackboard and Privacy layers provides for a more powerful means of dealing with specific domains of hierarchies and offers some synergies in hierarchy composition, such as activating and deactivating agents according to privacy settings or modifying the privacy settings through agents according to context.

Lastly, this work has proven its potential for successfully adapting to various types of environments and scenarios. Firstly, by **allowing problems of very different nature to be addressed**, such as controlling all the lights through a single switch (in an almost switchless environment), redirecting and synchronizing camera images according to context to create a virtually united distributed classroom, configuring the alert system of a security room, inferring location information to support and test other research or controlling the access rights to a working place, among others. Secondly, by **allowing for the smooth integration of new technologies in Intelligent environments**, either software (such as the University of Zaragoza's anthropomorphic figure, see Section 6.3.1),

hardware (such as the University of Calgary's Phidgets, see Section 6.3.2) and software–hardware systems (such as the University of Lancaster's steerable projection, see Section 6.3.3). Thirdly, by **allowing the integration with different UI** either designed for programmers or end–users (see Section 6.2). And finally, by **allowing for the coexistence and combination of novice, expert, programmer and research end–users**, by providing an easy and fast way to interact with and program the environment (as in the personal, working, teaching and researching environments of Section 6.1). The experienced gained over the course of this work has strengthened our initial beliefs that the synergy between human intelligence and the environment's capabilities can be triggered with a sound controlling mechanism.

7.1 Future Work

This work presents a combination of four different lines of research, some of which have been emphasized more than others, namely *Ambient Intelligence*, *Human Computer Interaction*, *Automatic Reasoning* and *Multi–agent systems*. All these lines of research are actively open at present and most can be studied and applied with more emphasis to the domain of this work.

First of all, the system presented in this work has been designed as a working system and therefore applied to different real environments. In this sense, we might extend our research to include new types of environments and users with special needs. Along these lines we are currently engaged in two Spanish projects that embrace particular and very different groups of users: elderly people with Alzheimer's and those afflicted by Down syndrome. We might evaluate the use of agents–rules in their particular scenarios, as well as the benefits of the magnetic poetry metaphor in these concrete groups and study how generic the metaphor is.

Language

Regarding the ECA–rule language, the limitations of the test environments have driven us to pay special attention to particular groups of events/conditions/actions. In this sense, different domains and automation scenarios will require studying other kinds of components. In particular we are studying how to deal with future and past conditions such as "if the previous time the door was open..." or "If yesterday at this time the door was open...". This timing issue is also present in conditions and actions relating to audio/video streams in which the use of expressions such as "the last 3 minutes of recorded video" will allow the user to achieve a complete multimedia control experience.

One of the main lines of research currently open is the extension of the set of comparators and operators of the grammar. In particular the study of operators

and comparators for sets (to take full advantage of the possibilities of *Wildcards* and *relations*) and the possible benefits of using arithmetic conditions in end–user oriented scenarios.

Finally, the potential of an event logic approach over an event algebra needs to be considered further. In this line of research, we are currently studying the potential of self–reference in Timers —the implications of allowing a Timer to launch itself as an action of one of its rules— and communication between Timers and their launching entities (i.e. a Timer being able to reference the Wildcard values of the rule that launched it or a Timer that can stop, restart or kill the Timer that launched it).

Agents

Regarding the multi–agent mechanism, there are a number of research lines with promising potential.

Firstly, in relation to the explanation mechanism of Section 5.3.1, research should be conducted on global explanation mechanisms, meaning that the user is able to ask the overall system for an explanation of an action. This requires a basic answering mechanism for the agent responsible for the action. Another more interesting approach is an "explanation on inaction" service, that is, how can the system answer a question of the type "Why didn't you turn on the lights right now?". Since no action took place, finding the agent responsible is more difficult and will require more communication mechanisms among agents than the previous service.

Also along the lines of autonomous agents, mobility and self–management should be considered. This work sets the basis for identifying breakdowns but there is no mechanism for the system to act upon that knowledge and restart the dead agent. In addition, at this point, agents are fixed to the machine in which they are launched. Given the changing nature of Intelligent Environments, in which people and resources may move to one place or another, computing devices can turn on and off and there are a multitude of small computers distributed among the environment, agents should be able to move across the network to keep running even under these changing conditions.

In addition, the potential of the *confidence factor* of actions (exposed in Section 5.3.2) set the basis for automatic learning, but no current mechanism is implemented for suggesting possible triggers or conditions that may disambiguate an ambiguous context (in which the action is successively right and wrong). Given the nature of Ambient Intelligence, in which the training examples are few and distributed in time, this kind of learning should be based on semantic heuristics (knowing that rules in an environment probably depend on variables of that same environment) in addition to pure pattern matching.

Finally, finding potential conflicts among different agents before the conflict

occurs is an interesting feature that will result in more competent systems. Thus, users could be advised of potential conflicts between their respective preferences in order to improve their hierarchies. This line of research is already open, using Software Engineering techniques in general and, in particular, studying the potentials of graph transformation tools to analyze the rules generated by the different users in order to check the model. This will probably require considering the implications of modeling the rules in the environment in one way or another. The benefits of this kind of intelligent data analysis approach to increase reliability have been already pointed out by Augusto, Nugent and McCullagh [5][4].

End–user Interfaces

Finally, regarding end–user programming, this work has presented the fundamental kernel language for an end–user programming system for Intelligent Environments, albeit with some basic UIs to ease the programming process. Nevertheless, a good end–user interaction/programming experience must count toward both a natural and easy programming paradigm and interface. Therefore, we should study other natural interaction paradigms such as oral interaction or tangible user interfaces, and how to overcome the problems stated in Section 6.2.2.

In addition to UI for creating rules, it is necessary to explore what kind of interfaces are best suited for users to manage their agent system. This includes some representation in the real world of the virtual agents present in an environment, a personal manager to create, delete or modify agents and a sharing mechanism to allow users to interchange/search behaviors.

Regarding the Magnet Poetry UI, a number of issues must be studied to allow the interface to be valid in larger and more complex scenarios. While some of these issues are being studied, the main lines of future work are the following: improving efficiency and effectiveness, and management.

Efficiency can be improved by providing search aids for easily finding the magnets with the desired concept. This includes a predictive text mechanism for finding or filtering tokens using a keyboard, and allowing the users to create their own tokes subsuming a combination of other tokens, for example.

Effectiveness, on the other hand, can be improved by enhancing the debugging mechanisms, such as by providing a more natural feedback of how well the system interpreted a rule.

Finally, exploring how to manage user–defined statements includes studying what kind of approaches are best suited to organize the user's preferences. A first alternative is to allow the users not only to create new behaviors, but to manage their complete set of agents and rules, moving rules from one agent to another or creating/deleting/modifying rules and agents from the same place. Conversely, a second alternative would be to allow the users to defer to the system the automatic organization of their rules. This alternative involves automatically

grouping rules according to some natural modularization concepts (such as what space or device they affect), as well as considering the modularization of other users with a similar set of rules. While this alternative limits the personalization options provided by the modularization mechanism, it will probably be preferred by truly novice users, while experienced users will still be able to use the plethora of tools provided by the system, for both managing (agents) and programming (the language).

7.2 Dissemination and contributions

This work has been featured in and produced a number of research publications, and has contributed to diverse research projects.

7.2.1 Publications

- Manuel García-Herranz, Xavier Alamán, and Pablo Haya. The intelligence of Intelligent Environments. Instinctive Computing International Workshop, Carnegie Mellon University, Pittsburgh, USA, June 15–16 2009. To be published as a book chapter in the LNAI State–of–the–Art Surveys.

- Manuel García-Herranz, Pablo Haya, Xavier Alamán. Easing the Smart Home: Translating human hierarchies to intelligent environments. In Joan Cabestany, Franciso Sandoval, Alberto Prieto, and Juan M. Corchado, editors, IWANN(1), volume 5517 of Lecture Notes in Computer Science, pp. 1529–1544. Springer, 2009

- Manuel García-Herranz, Pablo A. Haya, Germán Montoro, Abraham Esquivel, and Xavier Alamán. Easing the Smart Home: Semi-automatic Adaptation in Perceptive Environments. Journal Of Universal Computer Science (ISSN 0948–695x. Online edition: ISSN 0948-6968). 14 (9). 2008. pp. 1529–1544 JCR (0.315)

- Manuel García-Herranz, Pablo A. Haya, Xavier Alamán, and Pablo Martín. Easing the Smart Home: augmenting devices and defining scenarios. 2nd International Symposium on Ubiquitous Computing & Ambient Intelligence. Thomson, editorial. ISBN: 978-84-9732-605-6. Zaragora, Spain, 2007, pp. 67–74. **Best Paper Award**

- Manuel García-Herranz Pablo A. Haya, A. Esquivel, G. Montoro and X. Alaman. Semi-automation in Perceptive Environments: A rule-based agents proposal. VII Congreso Internacional de Interacción Persona-Ordenador. ISBN: 84-690-1613-X. (INTERACCION 2006). Puerto Llano, Spain. 2006, pp. 81–90.

- Manuel García-Herranz, Pablo A. Haya, G. Montoro, A. Esquivel, and X. Alamán. Adaptación automática de entornos activos mediante agentes basados en reglas. 2nd International Workshop on Ubiquitous Computing & Ambient Intelligence. ISBN: 84-6901744-6. Puertollano, Ciudad Real, Spain, 2006 pp. 189–204

- Javier Gómez, Germán Montoro, Pablo A. Haya, Manuel García-Herranz, Xavier Alamán. Easing the integration and communication in ambient intelligence. Accepted for publication in International Journal of Ambient Computing and Intelligence (IJACI). 2009

- Pablo Llinás, Germán Montoro, Manuel García-Herranz, Pablo Haya, and Xavier Alamán. Adaptive interfaces for people with special needs. In Sigeru Omatu, Miguel Rocha, José Bravo, Florentino Fernández Riverola, Emilio Corchado, Andrés Bustillo, and Juan M. Corchado, editors, IWANN(2), volume 5518 of Lecture Notes in Computer Science, pp. 772–779. Springer, 2009.

- A. Esquivel, P.A. Haya, M. García-Herranz, X. Alamán. A Proposal for Facilitating Privacy-Aware Applications in Active Environments. Advances in Soft Computing (ISSN 1615-3871). Vol. 51, 2009, pp. 312–320

- A. Esquivel, P.A. Haya, M. García-Herranz, X. Alamán. Harnessing the "Fair Trade" metaphor as privacy control in Ambient Intelligence. Ambient Intelligence Perspectives (ISBN 978-1-58603-946-2). P. Mikulecky et al. (Eds.). IOS Press. 2008, pp. 73–81.

- Abraham Esquivel, Pablo A. Haya, Manuel García-Herranz, Xavier Alamán. Managing Pervasive Environment Privacy Using the "fair trade" Metaphor. Lecture Notes in Computer Science (LNCS), ISSN 0302-9743 . Vol. 4806. 2007. pp. 804–813

- Germán Montoro, Manuel García-Herranz, Pablo A. Haya, Xavier Alamán, Daniel Brande, Sandra Baldassarri, Eva Cerezo y Francisco J. Seron. Integración de un agente virtual 3D en un entorno de inteligencia ambiental. 2nd International Symposium on Ubiquitous Computing & Ambient Intelligence (UCAMI'07). José Bravo, Xavier Alamán, editores. Thomson, editorial. ISBN: 978-84-9732-605-6. 2007 pp. 135–142.

- P.A. Haya, G. Montoro, A. Esquivel, M. García-Herranz and X. Alamán. Desarrollo de aplicaciones sensibles al contexto en Entornos Activos. 2nd International Workshop on Ubiquitous Computing & Ambient Intelligence. ISBN: 84-6901744-6. Puertollano, Ciudad Real, Spain. 2006. pp. 205–217

- Pablo A. Haya, Abraham Esquivel, Germán Montoro, Manuel García-Herranz, Xavier Alamán, Ramón Hervás, José Bravo. A Prototype of Context Awareness Architecture for Ambience Intelligence at Home. International Symposium on Intelligent Environments. ISBN: 1-59971-529-5. Cambridge, United Kingdom. 2006. pp. 49–55.

- Pablo A. Haya, Germán Montoro, Abraham Esquivel, Manuel García-Herranz and Xavier Alamán. A Mechanism for Solving Conflicts in Ambient Intelligent Environments. Journal Of Universal Computer Science (ISSN 0948-695x. Online edition: ISSN 0948-6968), 12 (3), 2006. 284–296. JCR (0.315)

7.2.2 Projects

HADA

Hipermedia Adaptativa para la atención a la Diversidad en entornos de inteligencia Ambiental (HADA) is a project supported by the Spanish Ministry of Education and Science (TIN2007-64718). The project is currently open (from 01/10/2007 to 31/09/2010), and its main goal is to develop technologies and tools to integrate Adaptive Hypermedia and Intelligent Environments to ease the use of new technologies for users with special needs, specifically the elderly and those with Down syndrome. http://hada.ii.uam.es/

ALIADO

ALzheimer Intelligent Ambient DOmótic system (ALIADO) is a project supported by the Spanish Ministry of Industry, Tourism and Commerce (TSI2008-020100-2008-296). It is a currently open project (from 01/01/2008 to 31/06/2009) in which several universities are involved (Universidad Autónoma de Madrid, Universidad de Castilla la Mancha and Universidad de Salamanca) along with some companies (CEDETEL and Tulecom Group S.L.). Its main goal is to develop technologies to support people with Alzheimer's disease.

Hesperia

Homeland sEcurity: tecnologíaS Para la sEguridad integRal en espacIos públicos e infrAestructuras (Hesperia) is a currently open project (from 2005 to 2009) supported by the Spanish Ministry of Industry, Tourism and Commerce and private companies in which the Universidad Autónoma de Matrid participates through the Cátedra UAM–Indra. Its main objective is the development of technologies for the security and operational control of infrastructures and public spaces (see Section 6.1.3). http://www.proyecto-hesperia.org/

Itech Calli

Itech Calli ("inside the house" in the Nahuatl language) was a project supported by the Universidad Autónoma de Madrid and the Grupo Santander Central Hispano, already finished (from 01/01/2006 to 31/12/2007). The research groups participating in this project were the Universidad Autónoma de Madrid, Instituto Tecnológico Superior Zacatecas Norte and Universidad Nacional del Centro de la Provincia de Buenos Aires. Its main goal was to deploy an Active Environment at the Technological Institute Superior North Zacatecas, its main focus being on teaching environments (see Section 6.1.2). http://itechcalli.ii.uam.es/

U-CAT

Ubiquitous Collaborative Adaptive Training (U-CAT) was a project supported by the Spanish Ministry of Science and Technology (TIN2004-03140). Currently finished (from 01/01/2005 to 31/12/2007), its main goal was to develop integrated environments to facilitate the conduct of educational activities in arbitrary places by using different physical devices. http://orestes.ii.uam.es/ucat/

Bibliography

[1] http://www.knx.org/. [cited at p. 6]

[2] Xavier Alamán, Ruben Cabello, Francisco Gómez-Arriba, Pablo A. Haya, Antonio Martínez, Javier Martínez, and Germán Montoro. Using context information to generate dynamic user interfaces. In *10th International Conference on Human-Computer Interaction, HCI International 2003*, Crete, Greece, June 22-27 2003. [cited at p. 6, 45, 59, 119]

[3] H.P. Alesso and C.F. Smith. *Developing semantic web services*. AK Peters, Ltd., 2005. [cited at p. 33]

[4] J.C. Augusto and C.D. Nugent. A new architecture for smart homes based on ADB and temporal reasoning. In *Proceedings of 2nd International Conference On Smart homes and health Telematic, ICOST2004), Assistive Technology Research Series*, pages 106–113. Citeseer, 2005. [cited at p. 150]

[5] Juan Carlos Augusto and Paul J. McCullagh. Ambient intelligence: Concepts and applications. *Comput. Sci. Inf. Syst*, 4(1):1–27, 2007. [cited at p. 150]

[6] Juan Carlos Augusto and Chris D. Nugent. The use of temporal reasoning and management of complex events in smart homes. In Ramon López de Mántaras and Lorenza Saitta, editors, *ECAI*, pages 778–782. IOS Press, 2004. [cited at p. 32, 60]

[7] Francisco J. Ballesteros, Enrique Soriano, Gorka Guardiola Muzquiz, and Katia Leal Algara. Plan B: Using files instead of middleware abstractions. *IEEE Pervasive Computing*, 6(3):58–65, 2007. [cited at p. 28]

[8] Y. Bar-Yam. Analyzing the effectiveness of social organizations using a quantitative scientific understanding of complexity and scale. *NECSI Technical Report*, May 2007. [cited at p. 105]

[9] J.E. Bardram. The java context awareness framework (jcaf)-a service infrastructure and programming framework for context-aware applications. *Pervasive Computing*, pages 98–115, 2005. [cited at p. 28]

[10] C. Beckmann and A. Dey. Siteview: Tangibly programming active environments with predictive visualization. In *Adjunct Proceedings of UbiComp*, pages 167–168, 2003. [cited at p. 40]

[11] G. M. Bierman and P. Sewell. Iota: A concurrent XML scripting language with applications to home area networking. Technical Report UCAM-CL-TR-557, University of Cambridge, Computer Laboratory, January 2003. [cited at p. 38]

[12] Urs Bischoff and Gerd Kortuem. Rulecaster: A programming system for wireless sensor networks. In Paul J. M. Havinga, Maria Eva Lijding, Nirvana Meratnia, and Maarten Wegdam, editors, *EuroSSC*, volume 4272 of *Lecture Notes in Computer Science*, pages 262–263. Springer, 2006. [cited at p. 30, 31, 45, 50]

[13] Alan F. Blackwell. First steps in programming: A rationale for attention investment models. In *HCC*, pages 2–10. IEEE Computer Society, 2002. [cited at p. 47]

[14] Alan F. Blackwell and Rob Hague. AutoHAN: An architecture for programming the home. In *HCC*, pages 150–157. IEEE Computer Society, 2001. [cited at p. 38]

[15] Oliver Brdiczka, Patrick Reignier, and James L. Crowley. Supervised learning of an abstract context model for an intelligent environment, smart objects and ambient intelligence. In *SOC-EUSAI 2005, Grenoble 2005*, 2005. [cited at p. 25, 50, 59]

[16] Rodney Brooks. The intelligent room project. In *Proceedings of the 2nd International Cognitive Technology Conference (CT'97)*, Aizu, Japan, 1997. [cited at p. 29]

[17] Yang Cai. Instinctive computing. In Thomas S. Huang, Anton Nijholt, Maja Pantic, and Alex Pentland, editors, *Artifical Intelligence for Human Computing*, volume 4451 of *Lecture Notes in Computer Science*, pages 17–46. Springer, 2007. [cited at p. 10]

[18] S. Chakravarthy, R. Elmasri, and A. Aslandogan. SNOOP EVENT SPECIFICATION: FORMALIZATION ALGORITHMS, AND IMPLEMENTATION USING INTERVAL-BASED SEMANTICS. [cited at p. 79]

[19] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 606–617, Santiago, Chile, 1994. [cited at p. 65, 77, 79, 81, 84]

[20] K. Cheverst, H.E. Byun, D. Fitton, C. Sas, C. Kray, and N. Villar. Exploring issues of user model transparency and proactive behaviour in an office environment control system. *User Modeling and User-Adapted Interaction*, 15(3):235–273, 2005. [cited at p. 36, 55, 61]

[21] Michael Coen, Luke Weisman, Kavita Thomas, and Marion Groh. A context sensitive natural language modality for an intelligent room, March 14 1999. [cited at p. 37]

[22] Michael H. Coen. Building brains for rooms: Designing distributed software agents. In *AAAI/IAAI*, pages 971–977, 1997. [cited at p. 31]

[23] D. J. Cook, M. Youngblood, E. Heierman, K. Gopalratnam, S. Rao, A. Litvin, and F. Khawaja. Mavhome: An agent-based smart home. In *Proceedings of the IEEE International Conference on Pervasive Computing and Communications*, pages 521–524, 2003. [cited at p. 49, 102]

[24] J.C. Cook, D.J. Augusto and V.R. Jakkula. Ambient Intelligence: Technologies, applications and opportunities. *Pervasive and Mobile Computing*, 2009. [cited at p. 7]

[25] Alexandra I. Cristea and Michael Verschoor. The LAG grammar for authoring the adaptive web. In *ITCC (1)*, pages 382–288. IEEE Computer Society, 2004. [cited at p. 65]

[26] James L. Crowley, Joëlle Coutaz, Gaëtan Rey, and Patrick Reignier. Perceptual components for context aware computing. In Gaetano Borriello and Lars Erik Holmquist, editors, *Ubicomp*, volume 2498 of *Lecture Notes in Computer Science*, pages 117–134. Springer, 2002. [cited at p. 25]

[27] Mihaly Csikszentmihalyi. *Flow: The Psychology of Optimal Experience*. Harper Perennial, New York, 1991. [cited at p. 13, 41, 48, 54, 55, 56, 177]

[28] S. Davidoff, M.K. Lee, J. Zimmerman, and AK Dey. Socially-aware requirements for a smart home. In *Procs of the International Symposium on Intelligent Environments*, pages 41–44, 2006. [cited at p. 7, 10, 48, 51]

[29] Scott Davidoff, Min Kyung Lee, Charles Yiu, John Zimmerman, and Anind K. Dey. Principles of smart home control. In Paul Dourish and Adrian Friday, editors, *Ubicomp*, volume 4206 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2006. [cited at p. 14, 52, 55, 56, 58, 105]

[30] Paul de Vrieze, Patrick van Bommel, and Theo P. van der Weide. A generic adaptivity model in adaptive hypermedia. In Paul De Bra and Wolfgang Nejdl, editors, *AH*, volume 3137 of *Lecture Notes in Computer Science*, pages 344–347. Springer, 2004. [cited at p. 65]

[31] K.S. Decker. Task environment centered simulation. *Simulating Organizations: Computational Models of Institutions and Groups. AAAI*, 1996. [cited at p. 23]

[32] Dey, Anind K., Raffay Hamid, Chris Beckmann, Ian Li, and Daniel Hsu. a CAPpella: programming by demonstration of context-aware applications. In *Proceedings of ACM CHI 2004 Conference on Human Factors in Computing Systems*, volume 1, pages 33–40, 2004. [cited at p. 35]

[33] A.K. Dey, T. Sohn, S. Streng, and J. Kodama. iCAP: Interactive prototyping of context-aware applications. *Lecture Notes in Computer Science*, 3968:254, 2006. [cited at p. 32, 40, 52, 59, 66]

[34] Anind K. Dey, Daniel Salber, and Gregory D. Abowd. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction (HCI) Journal*, 16(2-4):97–166, 2001. [cited at p. 28]

[35] Robert B. Doorenbos. Production matching for large learning systems. Technical Report CS-95-113, Carnegie Mellon University, School of Computer Science. [cited at p. 67]

[36] Abraham Esquivel, Pablo A. Haya, Manuel García-Herranz, and Xavier Alamán. Managing pervasive environment privacy using the f̈air tradeM̈etaphor. In *International Workshop on Pervasive Systems, PerSys 2007*, 2007. [cited at p. 57, 59, 72, 109, 113]

[37] S. Fine, Y. Singer, and N. Tishby. The hierarchical hidden markov model: Analysis and applications. *Machine Learning*, 31:32, 1998. [cited at p. 24]

[38] Charles Forgy. Rete: A fast algorithm for the many patterns/many objects match problem. *Artif. Intell*, 19(1):17–37, 1982. [cited at p. 29, 67]

[39] Sandra Baldassarri Francisco Serón and Eva Cerezo. Maxineppt: Using 3d virtual characters for natural interaction. In *II International Workshop on Ubiquitous Computing and Ambient Intelligence (wUCAmI'2006)*, Puertollano, Ciudad Real, Spain, November, 14 2006. [cited at p. 133, 134]

[40] S. Franklin and A. Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. *Lecture Notes in Computer Science*, 1193:21–??, 1997. [cited at p. 94]

[41] Ernest J. Friedman-Hill. *Jess, The Java Expert System Shell*. Sandia National Laboratories, Livermore, CA, USA, March 1998. Version 4.0. [cited at p. 29]

[42] Brian R. Gaines. Modeling and forecasting the information sciences. *Inf. Sci*, 57-58:3–22, 1991. [cited at p. 53, 54]

[43] Krzysztof Gajos. Rascal - a resource manager for multi agent systems in smart spaces,shop of central and eastern europe on multi-agent systems ceemas. Krakow, Poland, 2001. [cited at p. 37]

[44] Krzysztof Gajos, Harold Fox, and Howard Shrobe. End user empowerment in human centered pervasive computing, July 31 2002. [cited at p. 37]

[45] C. Le Gal, J. Martin, A. Lux, and J. L. Crowley. Smartoffice: Design of an intelligent environment. *IEEE Intelligent Systems*, 16(4):60–66, July-August 2001. [cited at p. 30]

[46] A. Galton. Eventualities. *The Handbook of Time and Temporal Reasoning in Artificial Intelligence*, 2004. [cited at p. 32]

[47] A. Galton and J.C. Augusto. Two approaches to event definition. *Lecture notes in computer science*, pages 547–556, 2002. [cited at p. 79]

[48] H. Gardner. *Frames of mind*. Basic Books New York, 1983. [cited at p. 10]

[49] H.W. Gellersen, A. Schmidt, and M. Beigl. Multi-sensor context-awareness in mobile devices and smart artifacts. *Mobile Networks and Applications*, 7(5):341–351, 2002. [cited at p. 141]

[50] Javier Gómez, Germán Montoro, and Pablo A. Haya. ifaces: Adaptative user interfaces for ambient intelligence. In *IADIS Multi Conference on Computer Science and Information Systems*, pages 133–141, Amsterdam, The Netherlands, 25–27 July 2008. IADIS Press. [cited at p. 6, 119]

[51] Guido Governatori, Michael J. Maher, Grigoris Antoniou, and David Billington. Argumentation semantics for defeasible logics. In *PRICAI*, pages 27–37, 2000. [cited at p. 65]

[52] S. Greenberg and C. Fitchett. Phidgets: Easy development of physical interfaces through physical widgets. In *Proceedings of the 14th Annual ACM Symposium on User Interface Software and Technology - ACM UIST'01*, pages 209–218, Orlando, Florida, November 11-14 2001. ACM Press. Best paper award. Earlier version as report 2001-686-09. [cited at p. 6, 135]

[53] Saul Greenberg. Toolkits and interface creativity. *Multimedia Tools Appl*, 32(2):139–159, 2007. [cited at p. 53, 54]

[54] Rob Hague. End-user programming in multiple languages. Technical report ucam-cl-tr-651, phd thesis, University of Cambridge, Computer Laboratory, October 2005. [cited at p. 38]

[55] Daniela Hall, Christophe Le Gal, Jérôme Martin, Olivier Chomat, and James L. Crowley. Magicboard: A contribution to an intelligent office environment. *Robotics and Autonomous Systems*, 35(3-4):211–220, 2001. [cited at p. 67]

[56] L. Hamill and R. Harper. Talking Intelligence: A Historical and Conceptual Exploration of Speech–based Human–machine Interaction in Smart Homes. In *International Symposium on Intelligent Environments*, pages 121–127, Cambridge, United Kingdom, November 11-14 2006. Microsoft Research. [cited at p. 49]

[57] Nicholas Hanssens, Ajay Kulkarni, Rattapoom Tuchida, and Tyler Horton. Building agent-based intelligent workspaces. In *International Conference on Internet Computing*, pages 675–681, 2002. [cited at p. 37]

[58] Pablo A. Haya. *Tratamiento de Información Contextual en Entornos Inteligentes*. PhD thesis, Universidad Autónoma de Madrid, 2006. [cited at p. 6]

[59] Pablo A. Haya, Abraham Esquivel, Germán Montoro, Manuel García-Herranz, Xavier Alamán, Ramón Hervás, and José Bravo. A prototype of context awareness architecture for ambience intelligence at home. In *International Symposium on Intelligent Environments*, pages 49–55, Cambridge, United Kingdom, 2006. Microsoft Research. [cited at p. 6, 31]

[60] Pablo A. Haya, Germán Montoro, and Xavier Alamán. A prototype of a context-based architecture for intelligent home environments. In *International Conference on Cooperative Information Systems (CoopIS 2004)*, volume 3290 of *Lecture Notes in Computer Science (LNCS)*, Larnaca, Cyprus, October 25-29 2004. [cited at p. 57, 58, 59, 107, 124]

[61] Pablo A. Haya, Germán Montoro, Xavier Alamán, Rubén Cabello, and Javier Martínez. Extending an xml environment definition language for spoken dialogue and web-based interfaces. In *In Developing User Interfaces with XML: Advances on User Interface Description Languages Workshop at AVI04*, Gallipoli, Italy, May 25 2004. [cited at p. 6]

[62] Pablo A. Haya, Germán Montoro, Abraham Esquivel, Manuel García-Herranz, and Xavier Alamán. A mechanism for solving conflicts in ambient intelligent environments. *Journal Of Universal Computer Science*, 12(3):284–296, 2006. [cited at p. 57, 63, 72, 108]

[63] Sumi Helal, William Mann, Hicham El-Zabadani, Youssef Kaddoura, and Erwin Jansen. The gator tech smart house: A programmable pervasive space. *IEEE Computer*, 38(3):50–60, March 2005. [cited at p. 28, 137]

[64] Jean-Michel Hoc and Anh Nguyen-Xuan. *Language Semantics, Mental Models and Analogy*, chapter 2.3, pages 139–156. Academic Press, 1990. [cited at p. 53]

[65] Stephen S. Intille. Designing a home of the future, July 11 2002. [cited at p. 34]

[66] Stephen S. Intille and Kent Larson. Designing and evaluating supportive technology, June 29 2003. [cited at p. 26, 34]

[67] C. Z. Janikow. Exemplar learning in fuzzy decision trees, August 14 1996. [cited at p. 36]

[68] N. R. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Journal of Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998. [cited at p. 94]

[69] Antonis C. Kakas and Pavlos Moraitis. Argumentation based decision making for autonomous agents. In *AAMAS*, pages 883–890. ACM, 2003. [cited at p. 105]

[70] Achilles Kameas, Stephen J. Bellis, Irene Mavrommati, Kieran Delaney, Martin Colley, and Anthony Pounds-Cornish. An architecture that treats everyday objects as communicating tangible components. In *PerCom*, page 115. IEEE Computer Society, 2003. [cited at p. 37]

[71] J. Kay, B. Kummerfeld, and P. Lauder. Managing private user models and shared personas. In *UM03 Workshop on User Modeling for Ubiquitous Computing*, pages 1–11, 2003. [cited at p. 36, 55]

[72] Cory K. Kidd, Robert Orr, Gregory D. Abowd, Christopher G. Atkenson, Irfan A. Essa, Blair MacIntyre, Elizabeth Mynatt, Thad E. Starner, and Wendy Newstetter. The aware home: A living laboratory for ubiquitous computing research. In *Proceedings of the Second International Workshop on Cooperative Buildings - CoBuild'99*, 1999. [cited at p. 7, 28, 29, 118]

[73] Klemmer, Scott R., Jack Li, James Lin, Landay, and James A. Papier-mache: toolkit support for tangible input. In *Proceedings of ACM CHI 2004 Conference on Human Factors in Computing Systems*, volume 1, pages 399–406, 2004. [cited at p. 40]

[74] Andrew Jensen Ko, Brad A. Myers, and Htet Htet Aung. Six learning barriers in end-user programming systems. In *VL/HCC*, pages 199–206. IEEE Computer Society, 2004. [cited at p. 47]

[75] A.A. Kulkarni. *A reactive behavioral system for the intelligent room*. PhD thesis, Massachusetts Institute of Technology, 2002. [cited at p. 29]

[76] Ajay Kulkarni. Design principles of a reactive behavioral system for the intelligent room, April 09 2002. [cited at p. 29, 37]

[77] Victor Lesser, Michael Atighetchi, Brett Benyo, Bryan Horling, Anita Raja, Régis Vincent, Thomas Wagner, Ping Xuan, and Shelley XQ. Zhang. The UMASS intelligent home project. In Oren Etzioni, Jörg P. Müller, and Jeffrey M. Bradshaw, editors, *Proceedings of the Third Annual Conference on Autonomous Agents (AGENTS-99)*, pages 291–298, New York, May 1–5 1999. ACM Press. [cited at p. 23]

[78] Pattie Maes. Agents that reduce work and information overload. *Communications of the ACM*, 37(7):31–40, July 1994. [cited at p. 9, 55, 56, 58, 107]

[79] Abraham H. Maslow. *Motivation and Personality*. Harper, New York, 1954. [cited at p. 105]

[80] A.H. Maslow. A theory of human motivation. *Twentieth Century Psychology: Recent Developments in Psychology*, page 22, 1946. [cited at p. 7]

[81] Irene Mavrommati, Achilles Kameas, and Panos Markopoulos. An editing tool that manages device associations in an in-home environment. *Personal and Ubiquitous Computing*, 8(3-4):255–263, 2004. [cited at p. 37]

[82] JD Mayer and P. Salovey. The intelligence of emotional intelligence. *Intelligence(Norwood)*, 17(4):433–442, 1993. [cited at p. 10]

[83] Marvin Minsky. *The Emotion Machine*. Simon & Schuster, New York, 2006. [cited at p. 10, 11, 66]

[84] David Molyneaux, Hans Gellersen, Gerd Kortuem, and Bernt Schiele. Cooperative augmentation of smart objects with projector-camera systems. In John Krumm, Gregory D. Abowd, Aruna Seneviratne, and Thomas Strang, editors, *UbiComp*, volume 4717 of *Lecture Notes in Computer Science*, pages 501–518. Springer, 2007. [cited at p. 80, 133, 141, 178]

[85] Germán Montoro. *Estudio e integración de un sistema de diálogos dinámico en un entorno inteligente*. PhD thesis, Universidad Autónoma de Madrid, 2005. [cited at p. 6]

[86] Germán Montoro, Xavier Alamán, and Pablo A. Haya. *Advances in Pervasive Computing*, chapter Spoken interaction in intelligent environments: a working system. Eds. Austrian Computer Society (OCG), 2004. [cited at p. 6, 119, 134]

[87] Germán Montoro, Xavier Alamán, and Pablo A. Haya. A plug and play spoken dialogue interface for smart environments. In *Fifth International Conference on Intelligent Text Processing and Computational Linguistics (CICLing'04)*, volume 2945 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, February 15-21 2004. [cited at p. 6]

[88] Germán Montoro, Pablo A. Haya, and Xavier Alamán. Context adaptive interaction with an automatically created spoken interface for intelligent environments. In *The 2004 IFIP International Conference on Intelligence in Communication Systems (INTELLCOMM 04)*, number 3283 in Lecture Notes in Computer Science (LNCS), Bangkok, Thailand, November 2004. [cited at p. 45, 59]

[89] Germán Montoro, Pablo A. Haya, Sandra Baldassarri, Eva Cerezo, and Francisco J. Serón. A study of the use of a virtual agent in an ambient intelligence environment. In Helmut Prendinger, James C. Lester, and Mitsuru Ishizuka, editors, *IVA*, volume 5208 of *Lecture Notes in Computer Science*, pages 520–521. Springer, 2008. [cited at p. 134]

[90] Michael Mozer, Lucky Vidmar, and Robert H. Dodier. The neurothermostat: Predictive optimal control of residential heating systems. In Michael Mozer, Michael I. Jordan, and Thomas Petsche, editors, *NIPS*, pages 953–959. MIT Press, 1996. [cited at p. 23]

[91] Michael M. Mozer. The neural network house: An environment that adapts to its inhabitants. In *Proceedings of the AAAI Spring Symposium on Intelligent Environments*. AAAI Press, 1998. [cited at p. 22, 49]

[92] Debra Mozer, Michael C., Dodier, Robert H., Anderson, Marc, Vidmar, Lucky, Cruickshank, Robert P III, Miller. The neural network house: An overview. 1994. [cited at p. 45]

[93] Brad A. Myers, John F. Pane, and Andy Ko. Natural programming languages and environments. *Commun. ACM*, 47(9):47–52, 2004. [cited at p. 12, 15, 50, 51, 53, 66, 67]

[94] Elizabeth D. Mynatt, Jim Rowan, Sarah Craighill, and Annie Jacobs. Digital family portraits: supporting peace of mind for extended family members. In *CHI '01: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 333–340, New York, NY, USA, 2001. ACM. [cited at p. 45]

[95] Bonnie A. Nardi. *A Small Matter of Programming: Perspectives on End User Computing*. MIT Press, Cambridge, MA, USA, 1993. [cited at p. 53]

[96] Mark W. Newman, Jana Z. Sedivy, Christine Neuwirth, W. Keith Edwards, Jason I. Hong, Shahram Izadi, Karen Marcelo, and Trevor F. Smith. Designing for serendipity: supporting end-user configuration of ubiquitous computing environments. In *Symposium on Designing Interactive Systems*, pages 147–156, 2002. [cited at p. 6, 9]

[97] M.W. Newman, T.F. Smith, and B.N. Schilit. Recipes for digital living. *Computer*, 39(2), 2006. [cited at p. 63]

[98] I. Nieto, J. Botía, and A. Gómez-Skarmeta. Information and hybrid architecture model of the ocp contextual information management system. 12(3):357–366, 2006. |http://www.jucs.org/jucs_12_3/information_and_hybrid_architecture—. [cited at p. 33]

[99] N. Oliver, A. Garg, and E. J. Horvitz. Layered representations for learning and inferring office activity from multiple sensory channels. *Computer Vision and Image Understanding*, 96(2):163–180, November 2004. [cited at p. 139]

[100] John F. Pane and Brad A. Myers. Tabular and textual methods for selecting objects from a group. In *VL*, pages 157–164, 2000. [cited at p. 40]

[101] Seymour Papert. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, New York, 1980. [cited at p. 12, 55, 56, 60, 66, 85]

[102] Adrian Paschke. ECA-ruleML: An approach combining ECA rules with temporal interval-based KR event/action logics and transactional update logics. *CoRR*, abs/cs/0610167, 2006. informal publication. [cited at p. 65, 77]

[103] Adrian Paschke. The reaction ruleML classification of the event / action / state processing and reasoning space. Technical report, November 10 2006. [cited at p. 76]

[104] N. W. Paton and O. Diaz. Active database systems. *ACM Computing Surveys*, 31(1):63–103, 1999. [cited at p. 68, 76, 77, 98]

[105] Brenton Asher Phillips. Metaglue :–a programming language for multi-agent systems. Master's thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1999. [cited at p. 29]

[106] Alexander Repenning and Andri Ioannidou. *What makes End-User Tick? 13 Design Guidelines*, chapter 4, pages 51–85. Human-Computer Interaction Series, Vol. 9. Springer-Verlag, 2006. [cited at p. 13, 54, 55, 56]

[107] Tom Rodden, Andy Crabtree, Terry Hemmings, Boriana Koleva, Jan Humble, Karl-Petter Åkesson, and Pär Hansson. Between the dazzle of a new building and its eventual corpse: assembling the ubiquitous home. In David Benyon, Paul Moody, Dan Gruen, and Irene McAra-McWilliam, editors, *Conference on Designing Interactive Systems*, pages 71–80. ACM, 2004. [cited at p. 41]

[108] Manuel Román, Christopher K. Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell, and Klara Nahrstedt. Gaia: A middleware infrastructure to enable active spaces. *IEEE Pervasive Computing*, pages 74–83, Oct-Dec 2002. [cited at p. 28]

[109] Bill Schilit, Norman Adams, and Roy Want. Context-aware computing applications. In *IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, US, 1994. [cited at p. 27]

[110] Albrecht Schmidt. Implicit human computer interaction through context. *Personal and Ubiquitous Computing*, 4(2/3), 2000. [cited at p. 31, 59]

[111] N. Stash, A. Cristea, and P. De Bra. Explicit intelligence in adaptive hypermedia: Generic adaptation languages for learning preferences and styles. In *Proc. of HT2005 CIAH Workshop, Salzburg, Austria*, 2005. [cited at p. 65]

[112] Michael Stonebraker, Anant Jhingran, Jeffrey Goh, and Spyros Potamianos. On rules, procedure, caching and views in data base systems. *SIGMOD Rec.*, 19(2):281–290, 1990. [cited at p. 72]

[113] Joo Geok Tan, Daqing Zhang, Xiaohang Wang, and Heng Seng Cheng. Enhancing semantic spaces with event-driven context interpretation. In Hans-Werner Gellersen, Roy Want, and Albrecht Schmidt, editors, *Pervasive*, volume 3468 of *Lecture Notes in Computer Science*, pages 80–97. Springer, 2005. [cited at p. 33, 59]

[114] A.S. Taylor. Intelligence in Context. In *International Symposium on Intelligent Environments*, pages 35–44, Cambridge (United Kingdom), 5-7 April 2006. Microsoft Research. [cited at p. 42, 104]

[115] Georgios Theocharous, Khashayar Rohanimanesh, and Sridhar Mahadevan. Learning hierarchical partially observable markov decision process models for robot navigation. In *ICRA*, pages 511–516. IEEE, 2001. [cited at p. 24]

[116] Khai N. Truong, Elaine M. Huang, and Gregory D. Abowd. CAMP: A magnetic poetry interface for end-user programming of capture applications for the home. In Nigel Davies, Elizabeth D. Mynatt, and Itiro Siio, editors, *Ubicomp*, volume 3205 of *Lecture Notes in Computer Science*, pages 143–160. Springer, 2004. [cited at p. 41, 52, 59, 129]

[117] K.N. Truong and G.D. Abowd. Inca: A software infrastructure to facilitate the construction and evolution of ubiquitous capture & access applications. In *In the Proceedings of Pervasive 2004: The Second International Conference on Pervasive Computing*, pages 140–157, Linz,Austria, April 2004. [cited at p. 42]

[118] Brygg Ullmer, Hiroshi Ishii, and Dylan Glas. mediablocks: Physical containers, transports, and controls for online media. In *SIGGRAPH*, pages 379–386, 1998. [cited at p. 38]

[119] Xiao Hang Wang, Da Qing Zhang, Tao Gu, and Hung Keng Pung. Ontology based context modeling and reasoning using owl. In *Proceedings of PerCom 2004*, pages 18–22, Orlando, FL, USA, march 2004. [cited at p. 33, 59]

[120] Roy Want, Andy Hopper, Veronica Falc ao, and Jonathan Gibbons. The Active Badge location system. *ACM Transactions on Information Systems*, 10(1):91–102, January 1992. [cited at p. 27]

[121] Roy Want, Bill Schilit, Norman Adams, Rich Gold, Karin Petersen, David Goldberg, John Ellis, and Mark Weiser. An overview of the parctab ubiquitous computing experiment. *IEEE Personal Communications*, 2(6):28–43, december 1995. [cited at p. 27]

[122] Mark Weiser. The computer for the 21st century. *Scientific American*, September 1991. [cited at p. 1, 2, 45, 48]

[123] Mark Weiser. Some computer science issues in Ubiquitous Computing. *Communications of ACM*, 36(7):75–84, July 1993. [cited at p. 1]

[124] G. Michael Youngblood, Diane J. Cook, and Lawrence B. Holder. Managing adaptive versatile environments. *Pervasive and Mobile Computing*, 1(4):373–403, 2005. [cited at p. 24, 26]

[125] T. Zhangt and B. Brügge. Empowering the user to build smart home applications. In *Toward A Human-Friendly Assistive Environment: ICOST'2004, 2nd International Conference on Smart Home and Health Telematics*, page 170. IOS Press, 2004. [cited at p. 67]

Appendices

Appendix A

Grammar

Table A.1: Rule–based language's grammar

rule	<triggerlist> '::' <conditionlist>? '⇒' <actionlist> ';'
triggerlist	<trigger> ('‖' <trigger>)*
trigger	element
conditionlist	<condition> ('&& '<condition>)*
condition	<element> <comparator> <element>
actionlist	<action> ('&& '<action>)*
action	<element> <operation> <element> <confidence>? | <timer>
confidence	'|' INT
timer	'TIMER' <timer_time> <concurrence>? '{' <timer_end_agent>? '}' '{' <rulelist>? '}'
timer_end_agent	<timer_end_stat>$^+$
timer_end_stat	timer_end_rule LINE_COMMENT COMMENT
timer_end_rule	(<conditionlist> '⇒')? <actionlist> ';'
timer_time	NAME
concurrence	INT
rulelist	<rule>+
comparator	EQUAL NOT_EQUAL GREATER SMALLER GREATER_OR_EQUAL SMALLER_OR_EQUAL
operation	ADD_PROPERTY ADD_RELATION ASSIGN ASSIGN_NOT CREATE_ENTITY MINUS PLUS REMOVE_RELATION
element	literal BB_ENTITY BB_ELEMENT
literal	NAME |INT

Table A.2: Lexicon

BB_ENTITY	NAME ':' NAME
BB_ELEMENT	NAME':'NAME':'NAME
NAME	(DIGIT* (LETTER|'_') ((LETTER|'_'|'.') |DIGIT)*'#'? (CHAR* (((' '|'\t') CHAR)* '*' '$'(DIGIT)+
LETTER	'a'..'z'|'A'..'Z'
CHAR	LETTER|DIGIT |'_'|'.'|'#'|'-'|'\\'|'|'|'/'|':'|'*'
INT	'0'..'9'+
EQUALS	'='
NOT_EQUALS	'!='
GREATER	'>'
SMALLER	'<'
GREATER_OR_EQUAL	'>='
SMALLER_OR_EQUAL	'<='
ADD_PROPERTY	'AP'
ADD_RELATION	'->'
REMOVE_RELATION	'-<'
ASSIGN	':='
ASSIGN_NOT	'=!'
CREATE_ENTITY	'CE'
PLUS : '+='	
MINUS : '-='	

Appendix B

Conclusiones

En este trabajo se ha presentado una estructura y lenguaje base para el control indirecto de Entornos Inteligentes. Tanto el lenguaje como la estructura han sido diseñados en semejanza a los procesos mentales naturales para expresar y organizar preferencias, persiguiendo un mecanismo de control a través del cual todos los elementos del entorno puedan ser usados y programados igualmente y de forma natural: **Acercando el modo de programación del entorno al modo en que los usuarios razonan acerca de su entorno**.

Este objetivo presenta dos problemas en los entornos actuales: los elementos computacionales **no comparten su información ni capacidades** ni **comparten sus mecanismos de interacción**. Esto es, a pesar de que diversos elementos del entorno son capaces de percibir y actuar en el entorno (como un termostato o un timbre), no comparten la información percibida ni sus capacidades de acción con el resto de los elementos del entorno. Adicionalmente, mientras que algunos de estos elementos pueden ser controlados y programados, cada uno de ellos presenta una forma y/o lugar de programación distinto del resto. La combinación de estos dos problemas redunda en una experiencia de control frustrante, volviéndose más frustrante a medida que crece el número de elementos activos, perceptivos, controlables o programables en el entorno.

Por otra parte, para diseñar un sistema que solucione estos problemas, se deben tener en cuenta otra clase de consideraciones. Analizadas en el Capítulo 3, estas consideraciones pueden ser resumidas en dos: la **heterogeneidad** de usuarios (preferencias, entornos y conocimientos) y su **multiplicidad** (la coexistencia de usuarios, preferencias y dominios de control).

La filosofía subyacente a este trabajo entiende que la inteligencia del entorno debe usarse para mejorar las capacidades de control de sus habitantes, en contraposición a otros enfoques que persiguen una inteligencia artificial capaz de gestionar de forma autónoma el entorno por el usuario (ver Capítulo 2). Así, este

trabajo se basa en un **paradigma de control automático**, a través del cual los usuarios son capaces de programar comportamientos que, de ahí en adelante, se producirán de forma automática, en contraposición con los **paradigmas de control autónomo** en los que el sistema decide qué es lo más adecuado para el usuario.

Finalmente, el trabajo desarrollado en esta tesis, persigue un **paradigma de programación independiente de la interfaz de programación**. Esto es, en lugar de centrarse en diseñar un interfaz de programación adecuada para el usuario, se persigue capturar la esencia de los mecanismos de programación naturales en un lenguaje que pueda ser traducido fácilmente a distintas interfaces de programación. De esta forma, mientras que diferentes interfaces de programación pueden ser creadas para cumplir determinados requisitos de interacción, el modelo subyacente de programación permanece invariable. Este tipo de sistema, que fija el modelo de programación, permite a los desarrolladores de interfaces centrarse exclusivamente en problemas de interacción

El mecanismo o sistema de programación se basa en dos componentes clave: un lenguaje de expresión basado en reglas ECA (Eventos Condiciones Acciones) y una estructura multiagente de modularización diseñadas, respectivamente, para hacer frente a los problemas de heterogeneidad y multiplicidad.

En relación con el lenguaje, el **lenguaje de reglas ECA** ha probado ser un **mecanismo natural de expresión**, como sugerían Myers y otros (ver Sección 3.3), especialmente para programar comportamientos reactivos que asocian una acción a un determinado contexto.

Por otra parte, al ser expresiones compresibles, proporcionan un base válida de **explicación** que, además de favorecer la confianza que el usuario deposita en su entorno (como sugieren múltiples autores, ver Sección 2), ha probado ser un **mecanismo de depuración** fundamental para mejorar la competencia del entorno. Adicionalmente, las reglas ECA, definidas por el usuario, proporcionan una base factible para el **aprendizaje automático** (ver Sección 5.3.2).

Sobre lo anterior, el hecho de disponer de un lenguaje base simplificado dotado de una serie de extensiones con las que expresar conceptos más complejos (como generalidad o tiempo), proporciona un mecanismo para aislar la complejidad del lenguaje, **haciendo frente a la relación inversa entre simplicidad y flexibilidad** propia de la mayor parte de sistemas de programación orientados al usuario. En este sentido, los usuarios sólo tienen que tratar con la complejidad estrictamente necesaria para resolver sus problemas. Por otra parte, el construir las extensiones complejas usando los conceptos y estructuras del lenguaje base proporciona un **curva de aprendizaje suave** en la que cada paso adelante facilita futuros avances.

Finalmente, de nuestra experiencia con los Timers (ver Sección 4.1.3) colegimos el potencial de usar una **lógica de eventos en lugar de un álgebra de**

eventos para construir eventos compuestos. Pese a que estas conclusiones poco tienen que ver con un enfoque directo en el usuario final, presenta algunas cuestiones interesantes en relación con la composición de eventos. En este sentido, los Timers permiten una definición de eventos compuestos por medio de procedimientos en contraposición con la declarativa de las álgebras de eventos (en las que un conjunto de eventos compuestos combinables como secuencia, disyunción o cierre son definidos) (ver Sección 4.1.4). Por otra parte, los eventos compuestos son codificados, con los Timers, junto con sus políticas de consumo, permitiendo combinaciones más flexibles de distintos eventos compuestos y políticas de consumo. Se han descrito las estructuras de Timers que codifican las álgebras de eventos y políticas de consumo más comunes en la literatura. En qué situaciones es apropiado renunciar a la facilidad que proporciona una solución declarativa y cómo de lejos puede llegar un mecanismo de descripción mediante procedimientos comparado con aquella son cuestiones abiertas que requieren un estudio más profundo. No obstante, los primeros pasos han sido marcados en este trabajo en torno a dos composiciones de eventos poco comunes: composición de eventos dependientes del contexto y políticas de consumo mixtas (ver Secciones 4.1.4 y 4.1.4).

En relación con el **sistema de modularización multiagente**, ha sido usado con éxito tanto para permitir una **evolución orgánica** del conjunto de preferencias del entorno como para **diseminar la responsabilidad** que el usuario pone en el sistema en diferentes entes independientes (en beneficio de un sistema más robusto y de una mayor confianza del usuario en el mismo) (ver Sección 5.1). Por otra parte, al representar los agentes en el modelo del mundo como una parte más (virtual) del entorno y dotarlos de un protocolo "are you alive?" o "¿estás vivo?" se abren las puertas a estrategias de recuperación automática con las que hacer frente a caídas inesperadas de los agentes.

Adicionalmente, el representar los agentes como una parte más del mundo, permite la creación de reglas que activen o desactiven agentes (esto es, conjuntos de preferencias) en relación con el contexto. En otras palabras, **construir jerarquías**. El etiquetar los agentes en el modelo del mundo con información contextual como para quién trabajan, dónde trabajan, que elementos son afectados por sus reglas o a qué tarea en particular se dedican, permite a los usuarios estructurar sus jerarquías de acuerdo a sus estructuras naturales. Por otra parte, un lenguaje basado en reglas proporciona un mecanismo natural para expresar las reglas de prioridad de una jerarquía y, al no imponer ningún grado de *interdependencia* y proporcionar diferentes grados de *escala* se garantiza que las estructuras de coordinación construidas con este sistema pueden tener **diversos grados posibles de complejidad** (como de hecho tienen las distintas estructuras sociales).

Finalmente, la combinación de las posibilidades de coordinación del mecan-

ismo multiagente con aquellas del modelo del mundo y la capa de privacidad, proporciona un potente mecanismo para trabajar con dominios jerárquicos específicos y ofrece diversas sinergias en la composición de jerarquías como activar o desactivar agentes de acuerdo a las preferencias de privacidad o modificar las preferencias de privacidad mediante agentes de acuerdo al contexto.

Por último, el presente trabajo ha demostrado su potencial de adaptación en diversos tipos de entornos y escenarios. En primer lugar, **permitiendo resolver problemas de muy diversa naturaleza** como controlar todas las luces de un entorno desde un solo interruptor, redirigir y sincronizar las imágenes de vídeo para unir virtualmente una clase distribuida, configurar un sistema de alertas en un entorno de seguridad, inferir información de localización con que probar otras tecnologías o controlar los derechos de acceso a un entorno laboral, entre otras. En segundo lugar, **permitiendo una integración fluida de nuevas tecnologías en Entornos Inteligentes**, tanto tecnologías software (como el avatar antropomorfo de la Universidad de Zaragoza, ver Sección 6.3.1), hardware (como los Phidgets de la Universidad de Calgary, ver Sección 6.3.2) o sistemas compuestos tanto de hardware como de software (como el sistema de proyección dirigible de la Universidad de Lancaster 6.3.3). En tercer lugar, **permitiendo su integración en distintas interfaces de usuario**, tanto diseñadas para programadores como para usuarios finales (ver Sección 6.2). Y, finalmente, **facilitando la programación a usuarios finales noveles, expertos, programadores y científicos**, proporcionando una manera fácil y rápida de interaccionar con y programar el entorno (como muestra la experiencia en entornos personales, laborales, educativos e investigadores descrita en la Sección 6.1). La experiencia ganada a lo largo del presente trabajo ha fortalecido en conjunto nuestras creencias iniciales en que la sinergia entre la inteligencia humana y las capacidades del entorno puede ser detonada proporcionando un mecanismo de control integral.

List of Figures

List of Tables