Using Stigmergy to Build Pervasive Computing Environments

Peter Barron

A thesis submitted to the University of Dublin, Trinity College

in fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

October 2005

Declaration

I, the undersigned, declare that this work has not previously been submitted to this or any other University, and that unless otherwise stated, it is entirely my own work.

Peter Barron

Dated: November 2, 2006

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Peter Barron

Dated: November 2, 2006

Acknowledgements

First and foremost I would like to express my deep gratitude to my supervisor, Prof. Vinny Cahill, for his support on this long journey. His encouragement and guidance have proven invaluable to me and I very much appreciate the time and effort he has given. I am lucky to have worked with him and I am thankful for the opportunities he has presented me.

In completing this research I was also privileged to work with a number of other academics in the Distributed System Group (DSG): to Dr. Simon Dobson my sincere thanks for starting me on this journey and for advising me during the early stages of my studies, to Dr. Siobhán Clarke my gratitude for sharing your expertise and for giving me the opportunity to work on the Carmen project. I am also grateful to Dr. Stefan Weber, Dr. Mads Haahr, Dr. René Meire, Dr. Jim Dowling, Dr. Christian Jensen and Stephen Barrett. Their advice and expertise has always been sought and greatly appreciated.

When I started in DSG I did not realise how much the people and the place would become so much a part of my life or how much I would enjoy it. As I reflect now I see what a unique and wonderful place it is. Of those in DSG there are a few who I'd like to give a special mention to: Tim Walsh, Raymond Cunningham, Andronikos Nedos, Elizabeth Gray, Barbara Hughes, Vinny Reynolds, Kulpreet Singh, Greg Biegel, Anthony Harrington and Jean-Marc Seigneur. I am sure I have omitted some but I thank you all for making it such an enjoyable experience.

To Ciara, your friendship and love has undoubtedly seen me through this Ph.D. You have kept me sane and have always managed to bring a smile to my face. I cannot express my gratitude enough, I only hope I can find the words some day.

Finally, to my family who without their love, encouragement, and unconditional support,

this thesis would simply not exist. To my Dad for his unwaivering belief in me, to my Mum for doing just about everything, and to my brother, Ian, who has always been there for me.

I thank you all,

Peter Barron
University of Dublin, Trinity College
October 2005

Abstract

Pervasive computing looks beyond the age of the personal computer to a time when everyday devices will be embedded with technology and connectivity. The goal of pervasive computing is to make such devices available throughout the physical environment to support people's usual activities with timely interventions without overwhelming them with inappropriate responses.

So far, a number of research efforts have investigated different approaches to managing the complexities of developing these types of environments with varying degrees of success. The difficulty still remains as to how to develop a pervasive computing environment that can support the integration and organisation of devices and applications in a spontaneous and robust manner. The problem is partly attributed to the highly dynamic and unpredictable nature of these types of environments, and is often further hampered by the limited resources found on devices.

The technology for pervasive computing is reaching a point where it is possible to convert may everyday environments into interactive spaces. Typically, these spaces have been designed from the ground up to support the anticipated needs of users, and are usually pre-installed and maintained over the period during which they are in use. Conceptually, these efforts are centralised in their approach, in that, efforts are focused around coordinating the resources of a particular geographical location in meeting the demands of users. However, in the future, the construction of pervasive computing environments is more likely to evolve accidentally from physical spaces as technology is incorporated into the space over time. This suggests there is a need to support the assembly of pervasive computing environments in a more ad-hoc fashion.

To address this point this thesis presents a highly-decentralised method of organising the

components of a pervasive computing environment, supporting spontaneous interaction between entities and providing robust system-wide behavior. The inspiration for this work stems from nature and the observations made by the French biologist Grassé on how social insects coordinate their actions using indirect communication via the environment, a phenomenon that has become known as stigmergy. In the stigmergic approach there are fewer dependences between entities allowing for the incremental construction and improvement of solutions without adversely effecting the rest of the pervasive computing environment. This thesis encapsulates this approach in a model that is used to underpin a framework for pervasive computing.

A prototypical implementation of the model is provided by Cocoa. Cocoa supports the use of stigmergy to build self-coordinating environments that promote the autonomy of entities. Designed to both support and complement the use of stigmergy, the framework employs a distributed architecture organised in a peer-to-peer fashion. To ease the implementation and deployment of entities Cocoa supports a programming abstraction encapsulated in a high-level scripting language. The scripting language exploits the methodologies used by social insects to construct a society of autonomous entities capable of responding to the environment in a stigmergic manner.

In order to validate the contribution of the thesis a select number of application scenarios from a range of different domains have been implemented using the Cocoa framework. The evaluation is used to demonstrate how a model based on stigmergy can be used to provide a highly-decentralised method of organising the components of a pervasive computing environment. In addition, the evaluation shows how such an approach can support the spontaneous interactions of autonomous entities and provide robust system-wide behavior.

Publications Related to this Ph.D.

Peter Barron and Vinny Cahill. Using Stigmergy to coordinate pervasive computing environments. *In Sixth IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'04).* pages 62-71, December 2004.

Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

Advances in technology are allowing pervasive computing to look beyond the realm of the personal computer (PC) to a time when everyday devices will be embedded with technology and connectivity. It is expected that large numbers of these devices will populate environments to such an extent that the physical and computational infrastructures will become so integrated that they will be transformed into spaces capable of supporting the activities of those who use them. The goal for pervasive computing is to ensure that the assistance provided is done in a timely manner without overwhelming users with inappropriate responses.

Pervasive computing has been used in a wide variety of areas including education where it has been used to support students in attending lectures [1, 16], and in offices to assist workers in meetings [78, 64] or in group collaborative sessions [153]. It has also been used in scientific laboratories to support the work of scientists [60, 62], and in the home to ensure the efficient usage of resources [107]. In addition, pervasive computing has also been used to support the elderly in the home [111] and to guide tourists [33].

The technology for pervasive computing is reaching a point where it is possible to convert many everyday environments into interactive spaces capable of supporting the activities of users. However, to develop a pervasive computing environment requires the consideration of new and alternative approaches to system design as traditional methods of development do not fully address the requirements of pervasive computing. For pervasive computing it is necessary to develop methods of supporting the integration and organisation of devices

and applications in a spontaneous and robust manner. This task is made considerably more difficult by the highly-dynamic and unpredictable nature of these types of environments and the limited resources often found on devices.

The research presented in this thesis addresses some of these issues. Firstly, the thesis proposes an alternative method of constructing pervasive computing environments based on the swarm intelligence technique of stigmergy [58]. Secondly, the thesis uses the concept of stigmergy to describe a highly decentralised method of organising the components of a pervasive computing environment that supports spontaneous interaction between entities and provides robust system-wide behavior. Using this approach the thesis presents a framework that supports the use of stigmergy in building self-coordinating environments that promote the autonomy of entities. Lastly, the thesis describes a programming abstraction encapsulated in a high-level scripting language for developing pervasive computing applications.

The proposed approach seeks to establish a method for assembling pervasive computing environments in a more ad-hoc fashion. The objective is to allow pervasive computing environments to evolve incrementally from physical spaces as the technology is incorporated into the space over time. This differs from current approaches where pervasive computing environments have typically been designed from the ground up to support the anticipated needs of users, and are usually pre-installed and maintained for the duration that they are in use. Conceptually, these efforts are centralised in their approach, in that, their focus is on coordinating the resources of a specific geographical location in meeting the demands of users. However, the future of pervasive computing lies in allowing physical spaces to evolve into pervasive computing environments with the inclusion of technology over time. The approach proposed in this thesis seeks to support such an approach.

This chapter begins with an introduction to the area of pervasive computing describing the general concepts and motivation for pervasive computing, and outlining the technologies that have spurred this area of research. The chapter continues by introducing the concepts of context and stigmergy and, following this, highlights the aims of the research presented here and outlines the methodology used in conducting it. To conclude, the chapter presents the main research contributions made by this work.

1.1 Pervasive Computing

When Mark Weiser wrote his seminal paper on ubiquitous computing[1] [167] he recognised that the world of computing was moving towards an era where everyday devices would be embedded with technology and connectivity. He realised that the widespread proliferation of these devices would require a change in how people use and interact with computers. In [168] Weiser wrote:

> "Ubiquitous computing is the method of enhancing computer use by making many computers available through the physical environment, but making them effectively invisible to the user" [168]

The statement indicates that the objective of ubiquitous computing is to transform physical spaces into interactive environments capable of reacting to, and meeting the needs of those who occupy them. However, the statement also illustrates Weiser's belief that for us to interact with hundreds of nearby wirelessly interconnected computers it is necessary for them to disappear from our awareness so that they may truly become an integral part of our daily lives.

Underlying this new wave of computing are advances in technology that have allowed computational devices to populate the environment around us in ever greater numbers. It is now reaching a point where the quantitative relationship between humans and computers has shifted so significantly from the time when many users shared only a single computer to a situation where many computers now share the attention of only a single person. The increasing number of devices in our environment provides the initial infrastructure that makes pervasive computing a possibility.

However, the vision proposed by Weiser is more than just the quantitative relationship between humans and computers, it also describes how an environment composed of hundreds of these devices are to interact with those who use it. The aim is to provide users with timely interventions that assist them with their current task without overwhelming them

[1] Ubiquitous computing has also become known as pervasive computing and in this thesis both terms are used interchangeability.

with inappropriate responses. Weiser describes this as calm computing [166] where technology requires little or no attention from users but is just there ready to be used when required.

Not surprisingly, the concept poses a number of substantial research challenges: integrating devices into an environment so they may sense and interact appropriately with users; supporting the spontaneous interoperability between devices to ensure their seamless integration into the environment; coordinating the behavior of devices to ensure a coherent environment can form; providing a system that can scale to the large collection of devices that are expected to be interwoven into the environment; ensuring a system behaves robustly in the presence of frequent failures of devices; supporting the high degree of mobility that is inevitable with users moving objects from one location to another; and maintaining the privacy of users who use the environment.

The task of accomplishing these challenges is made considerably more difficult by the general dynamic and unpredictable nature seen in these types of environments, and the often limited resources found on devices.

1.2 Enabling Technologies

The development of pervasive computing environments would not be possible without a number of key enabling technologies. Sensors have allowed computational devices to observe their surrounding environment and react to it. The availability of wireless communication has allowed devices to be distributed across the environment in a more ubiquitous manner. The abundance of small, cheap, portable devices has provided the backbone to this new era of computing. The work presented in this thesis is technology independent but assumes an underlying infrastructure capable of supporting a pervasive computing environment. This section therefore provides an overview of these technologies.

1.2.1 Sensors

To disappear into the background of society computers must first become part of it. This they achieve by sensing their surrounding environment. The information obtained allows

devices to adapt their behavior to meet the current situation. The process ensures that devices will always act appropriately in supporting the actions of users in their immediate environment. The increased variety of sensing technologies available at reasonable prices has allowed everyday objects to be embedded with these technologies allowing them to become part of society. For example, Point Six [147] provides a number of wireless sensors capable of measuring air temperature, humidity, and pressure. Their sensors run on batteries with a lifespan of up to 5 years and transmit data to receivers that can be up to 180 meters away.

The current generation of pervasive computing systems uses a wide variety of sensors, some of which are basic off-the-shelf components adapted for pervasive computing while others have been specifically designed to operate within these kinds of environments. For example, the Adaptive House [108] uses a number of sensors embedded into a house to detect the ambient illumination, room temperature, sound level, and motion of users to predict the inhabitants needs and to manage energy usage in a more efficient way. AT&T's sentient computing project [2, 65] has developed a location system called BAT [165]. The BAT system uses a combination of radio frequencies (RF) and ultrasound signals to provide very accurate location information in three dimensions. Transmitter devices, known as bats, are attached to objects or given to users. The signals transmitted are received by devices placed in the ceiling. Using the information from three of these devices, the location of a bat device can be triangulated to high degree of accuracy. AT&T have used the system to build a series of location-aware applications. For example, applications that followed users from one computer to another, or a service that routes telephone calls to the nearest phone beside a person. The Tea project [142] has attached photo-diodes, accelerometers, passive IR sensors, plus temperature and pressure sensors to a mobile phone, so that the device can change its behavior to suit the situation it is in.

While the advances in sensing technologies have allowed pervasive computing applications to gain a better understanding of the surrounding environment there are issues that arise from their use in these types of environments. The accuracy of sensors can vary considerably leading to a degree of uncertainty in the reliability of the readings obtained from sensors. Often pervasive computing systems tackle the issue through the use of sensor fusion techniques to

provide both more reliable and higher-level information on the environment. This derived information is generally described in pervasive computing as context information [42]. An introduction to context information is provide in section 1.3.

1.2.2 Communications

Wireless communication is playing an integral part in the development of pervasive computing as it encourages the ubiquitous nature of these types of environments. It ensures devices can retrieve information and coordinate their actions at any point without the restriction of having to be physically connected to a network. It allows devices to move through an environment and to be located in places that would not normally be covered by a wired network. A number of wireless technologies are used, some of which have been developed for generic use while others have been specifically designed for pervasive computing.

In projects such as Aura [56], Gaia [134] and the Stanford Interactive Workspace Project [78] the IEEE 802.11 [73] standard has been used in combination with a wired Ethernet network to provide the underlying communication network for the environment. IEEE 802.11 is a wireless network standard that can typically span a building, or campus of up to a few kilometers is size. The standard is able to operate in either of two modes; an infrastructural mode that relies on base stations to mediate communication between nodes and an ad-hoc mode which allows peer communication between nodes. When used in combination with ad-hoc routing protocols, such as AODV [119], the ad-hoc mode can provide full multi-hop communication between nodes. The nominal bandwidth of the IEEE 802.11b standard is 11Mbps, though the newer IEEE 802.11g standard can achieve 56Mbps.

In contrast to IEEE 802.11, a set of wireless technologies support short-range communication; generally in the range of 10s of meters. Bluetooth is one of the main technologies to be used at this range. It is normally used to connect and exchange information between personal devices such as personal digital assistants (PDAs), mobile phones, laptops, personal computers, printers, and digital cameras. Its normal bandwidth is 1Mbps. Bluetooth is developed by a consortium of interested parties that include Ericsson, Sony, IBM, Intel, Nokia and Toshiba. IrDA is another technology used in these types of networks. It uses infra-red

signals to perform short-range line of sight communications between devices and is used in a similar fashion to bluetooth. A number of wireless networks have been designed specifically for severely energy constrained devices where the consumption of power has to be kept to a minimum. Normally these type of networks are used within sensors networks [82] or for connecting devices such as those developed by the Smart-Its project [72].

While wireless communication networks encourage the ubiquitous nature of pervasive computing, they do increase the complexity of building these types of environments. Unlike traditional wired networks, which tend to be static, the nodes forming a wireless network are extremely dynamic within a pervasive computing environment. These nodes may have varying degrees of mobility but they are continuously moving through the environment and may enter or leave a network for any number of reasons. For example, a mobile node may move out of range of other nodes causing a partition in the network or a node may fail due to the lack of power or a fault within the device. Any pervasive computing system has to assume that failure and disconnected operations are the norm and not the exception and so must develop appropriate mechanisms for dealing with it.

1.2.3 Devices

The abundance of small, cheap, portable computing devices is the backbone to Weiser's vision of pervasive computing. It is these devices that provide the computational infrastructure for transforming the environment into an interactive space capable of supporting the activities of those who inhabit it. The current generation of pervasive computing environments use a wide selection of devices to accomplish this some of which are freely available from vendors while others have been specifically designed to be embedded into everyday objects.

In one of the first pervasive computing environments [164] Want et al. developed and experimented with three different types of devices: ParcTabs, ParcPads, and Liveboards. ParcTabs were small hand-held devices that provided an information doorway to users. ParcPads were larger notebook-sized devices and Liveboards were yard-sized interactive screens that could be placed in offices or at other locations within the Xerox PARC facility. Together these provided the computational infrastructure for the environment and the experiments

performed by Want et al.

In more recent projects such as the Stanford Interactive Workspace Project [78] or the Intelligent Room [64], a combination of workstations, notebooks, tablet PCs, and PDAs are used to provide the infrastructure for the environment. The devices that are in these projects come in a range of sizes and would typically incorporate wireless technologies such as IEEE 802.11, Bluetooth, or IrDA and may utilise touch-sensitive screens and stylus for user input instead of using a keyboard. Some of the newer generation of PDAs also include a number of sensors, including biometric sensors, embedded into the device to determine light intensity, location, and to read finger prints.

In contrast, the TEA project [142] uses mobile phones to support the activities of users. The mobile phones incorporates a number of sensors that can be used to adapt the behavior of the phone. For instance, in a noisy environment the phone changes its profile to increase the volume of the ring tone used. In general, mobile phones provide an ideal platform for pervasive computing due to there widespread use in society. The next generation of phones now integrate personal information management and mobile phone capabilities on the same device. In doing so, they provide a very flexible platform for developing pervasive computing applications.

Another project, called Smart-Its [72], has developed small devices of only a few centimeters in size that can be embedded into everyday objects. They incorporate a number of sensors and wireless communication and are designed to augment and interconnect artifacts within a pervasive computing environment. So far, they have been used to augment chemical containers to detect and alert potential hazardous situations [155], embedded into tables to track activity [143], and connected to skis to determine the performance of skiers [101, 102].

A similar project to Smart-Its is the Smart Dust project [82]. The Smart Dust project has developed an extremely small device called a Mote. It features a range of on-board sensors and has an integrated wireless network capability. The Smart Dust project has also developed specialised protocol stack for communication to allow the ad-hoc routing through large networks of Motes. Originally developed for wireless ad-hoc sensor networks Motes are now increasing being used in pervasive computing to augment everyday objects.

The progress of technology is reaching a point where computational devices are getting smaller and more numerous. It is possible to embedded large collections of these devices into our surrounding environment without us even knowing of their existence. While these devices provide the backbone for pervasive computing, a variety of issues arise from their wide spread proliferation within the environment. Unlike personal computers, which tend to be static and fairly robust, the devices forming a pervasive computing environment are extremely dynamic. Typically, they are small devices with limited resources for power, computation, communication, and storage. They also have varying degrees of mobility that allow them to move through the environment. While exepcted advances in storage capabilities will partly alivate this problem the limitations and tradeoffs between power, computation, and communication are anticipated to remain for the foreseeable future. A pervasive computing system must assume that due to their mobility and limited resources that failure will be common place and so must create mechanisms to ensure the robust behavior of the environment can be maintained.

1.3 An Introduction to Context

An important aspect of meeting Weiser's vision for calm computing [166] is the ability for systems to observe and understand what is occurring within the environment and to use this information to adapt their behavior to suit the situation. This is an important requirement of pervasive computing and is necessary for any system if it is blend seamlessly into the background and to determine correctly how it is to interact with users. The use of context, i.e. the information that is derived from data retrieved from sensors embedded in the environment, has provided the means of accomplishing this.

1.3.1 Context

Schilit and Theimer [139] define context as a "collection of nearby people and objects, as well as the changes to those objects over time". According to Schilit et al. [138] the important aspects of context are: where you are, who you are with, and what resources are nearby. They

also note that context is more than just the location of users, but includes other information such as lighting, noise levels, social situations. For Brown et al. [18], a user's context is determined by a collection of information such as their location, the season, temperature. Pascoe [118] defines context to be the subset of physical and conceptual states of interest to a particular entity. However, as Dey et al. [42] points out many of the above definitions are too specific. They observe that context is about the whole situation relevant to an application and define context as:

> "Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves." [42]

Definitions of context vary considerably within the research community. However, some key concepts can be drawn from the literature. Namely, that context describes the *state* of the environment that is *relevant* to an *entity*. Where the *identity* of the entity, the *time*, the *location* of the entity, and the *task* being performed play a primary role in describing an *entity's* context.

1.3.2 Context-Awareness

Dey et al. [42] define context-awareness as the use of context to provide relevant information and/or service to the user. Brown et al. [18] define context-aware applications as those that change their behavior according to the user's context. Schilit et al. [138] define it as the ability of software to adapt to the location of use, nearby people, and devices. There are a number of interpretations of context-awareness though the general consensus is that a system is context-aware if it uses or adapts to the context information derived from the surrounding environment.

The majority of pervasive computing systems use and adapt to context in a number of ways. For instance, they reconfigure systems to handle the changing availability of resources, react to users, and adapt the behavior of a system to meet the requirements of users. However,

there are a number of issues with using context information in pervasive computing. For instance, there is a level of uncertainty with the accuracy of the context information being provided due to the inherent inaccuracies of sensors. Another issue arises over the timeliness and frequency in which the context information is updated by the system to maintain the freshness of the information in use. There is also a question of how best to model the context information used to represent the environment in the system. While problems do exist with the use of context information, its use is what allows systems to meet Weiser's vision for pervasive computing.

1.4 Swarm Intelligence

The potential of simple behaving entities was first noted by biologists in their observations of colonies of social insects. They noticed that through local interactions of individuals, a colony of insects could produce complex collective behaviors at the colony level. These type of behaviors are also seen in flocking birds and shoals of fish and are considered to be highly distributed and largely self-organising. The mechanisms used to organise these types of systems and the collective behavior that emerges from them has become known as swarm intelligence. Bonabeaus et al. [12] describes swarm intelligence as the attempt to design algorithms or distributed problem-solving devices inspired by the collective behavior of social insect colonies and other animal societies. The ability of these simple behaving entities to produce complex behaviors has not gone unnoticed and has provided the inspiration for several research initiatives [47, 45, 46, 95, 23, 83] that used the same coordination mechanisms to solve a range of computer-related problems. One of the coordination mechanisms used is known as stigmergy.

1.4.1 Stigmergy

In 1959, the French biologist, Grassé observed that social insects could co-ordinate their actions through the environment without having to directly communicate with each other. They do this using a phenomenon known as *stigmergy* [58]. He also noticed that the local interac-

tions between insects resulted in the emergence of colony-wide behavior. Holland et al. [71] showed that stigmergy provides a mechanism that allows the environment to structure itself through the activities of the entities within the environment. The state of the environment, and the current distribution of entities within it, determines how the environment and the entities will change in the future. This approach provides a robust, self-organising environment, which allows entities to coordinate their behavior in a highly decentralised manner. It is important to stress that individual entities have no particular problem solving knowledge, and that coordinated behavior emerges due to the actions of the society. It also worth noting that while no direct communication is used between individual entities, communication is still maintained through the medium of the environment.

The trail-laying and trail-following used by many species of ants [12, 41] when foraging for food is a classical example of the use of stigmergy in nature. Ants deposit pheromones on their way back from a food source. Foraging ants follow such trails. The process has been shown to be self-organising [41] and capable of optimizing on the shortest path to the food source [57]. The nest building of social wasps [160] is another example of stigmergy used within nature. Nests are built up from wood fibers and plant hairs and cemented together with salivary secretions. These are then moulded by the wasp to form the different parts of the nest. Wasps coordinate the construction of a nest by each individual observing the local structure of the nest and deciding where to build the next part of the nest. Another example is the corpse gathering behavior seen in some species of ants. Worker ants pick up corpses in the nest and drop them in locations of higher concentrations to form piles of corpses in a process which acts to clean the nest.

1.4.2 Harnessing the Principles of Stigmergy

The idea of simple insects, with little memory or ability to exhibit any real intelligence, maps well to pervasive computing where small devices with limited resources are spread across the environment. The large number of devices expected to be deployed into our society matches the scale at which these colonies of social insects work. The constant interaction between devices of a pervasive computing environment also ties in neatly with how social insects

interact with each other. While the environment social insects use to coordinate their actions can be represented in a pervasive computing system by the context information derived from sensors.

The indirect communication mechanisms harnessed by social insects provide a means of decoupling devices and applications. Having fewer dependencies between components allows the overall system to be less fragile and more robust to disturbances in the environment. The system can grow organically and decay gracefully with the environment, as new devices are added and old ones upgraded or removed, without having an adverse effect on the overall system. The spontaneous interaction of devices and applications can be achieved as communication is realised through the common medium of the environment. The use of stigmergy allows us to harness the robust, self-organising, coordinating mechanisms of social insects, which are all desirable attributes for pervasive computing.

The thesis proposes to use the principles of stigmergy to create a pervasive computing environment, where context information from surrounding entities and the environment provide a common medium for an indirect communication mechanism to be used by entities. The social insects observed by Grassé are represented as entities within the pervasive computing environment. An entity is a person, place, or object as defined by Dey [42]. Co-ordinated behavior arises from entities observing their environment and reacting to the received context information according to a set of simple rules.

1.5 Research Scope, Aims and Methodology

The objective of the research presented in this thesis is to assess the use of stigmergy in supporting the development of pervasive computing systems. More specifically, the objective is to mimic the methodologies used by social insects to construct a society of autonomous entities capable of supporting spontaneous interaction between entities and providing robust system-wide behavior for a pervasive computing environment. The primary research aims are therefore to:

- Investigate the issues relating to the development and deployment of pervasive comput-

ing systems.

- Understand the implications of using techniques based on the phenomena of stigmergy to construct pervasive computing environments.
- Develop a model based on these techniques for developing pervasive computing environments.
- Use the model to develop a framework that supports the use of stigmergy to build self-coordinating environments that promote the autonomy of entities.
- Evaluate the effectiveness of using such an approach by developing a series of prototype scenarios to access the implications of using these techniques.

The methods used in conducting the research presented use a rigorous survey of the literature and available information, modeling of techniques, the design and implementation of a prototype to indicate the feasibility of such an approach, and an evaluation to understand the implications of using the proposed techniques.

1.6 Research Contribution

The work presented in this thesis has focused primarily on the investigation of techniques based on stigmergy in the design and implementation of pervasive computing environments. Consequently, the main contributions of the thesis can be summarised as follows:

- An overview of pervasive computing systems with respect to the integration and organisation of devices and applications within these types of the environments.
- A highly decentralized method for organising components of a pervasive computing environment that supports spontaneous interaction between entities and provides robust system-wide behavior.
- A framework, called Cocoa, that supports the use of techniques based on the phenomena of stigmergy to build self-coordinating environments which promote the autonomy of entities.

- A programming abstraction encapsulated in a high-level scripting language that generalises the methodologies used by social insects to construct a society of autonomous entities capable of responding to the environment in a stigmergic manner.

1.7 Thesis Outline

The following is an outline of the remainder of the thesis. Chapter 2 reviews Weiser's vision of ubiquitous computing and investigates the challenges posed by the area of research. The chapter then continues to examine a variety of relevant research projects and reviews their efforts in overcoming the challenges of ubiquitous computing. Chapter 3 presents a set of requirements that are used to influence the design and implementation of the framework. The requirements are drawn from an analysis of the related work presented in chapter 2. The chapter also introduces the natural phenomenon of stigmergy and investigates how it can be used to address the requirements. From this analysis, the chapter develops a model that the framework can support. Chapter 4 presents a programming abstraction in the form of a high-level scripting language that can be used to develop pervasive computing applications. Chapter 5 presents a prototypical implementation of the stigmergic model described in chapter 3 and of the scripting language described in chapter 4. To evaluate the effectiveness of the framework chapter 6 discusses a select number of application scenarios that demonstrate the use of the framework in development of pervasive computing environments.

Chapter 2

Background and Related Work

In order to set the scene for the thesis and provide necessary background for the work to be presented here, this chapter examines a number of research projects and reviews their efforts to overcome the hurdles posed by pervasive computing. Particular attention is paid to understanding the mechanisms used to integrate devices and applications into pervasive computing environments and to understand how they are organised within it. The chapter also notes the methodology used in their construction and the methods employed in developing applications for them. The aim is to gain a better understanding of the complexities of building pervasive computing environments and to develop an insight into how the current state of the art might be improved. The chapter begins with an introduction to pervasive computing.

2.1 Pervasive Computing

In 1991 Mark Weiser wrote his seminal paper [167] on ubiquitous computing. It was the start of what has become an alternative vision for the use of computers in society. Wieser had realised that, as the number of computers increase within society, it would become increasingly more difficult to use them in any meaningful way. He believed that it was necessary to move away from the conventional concept of the personal computer to an approach where physical spaces would be made up of computational devices, that would be integrated into everyday objects

and embedded with technology and connectivity. In this world, people would be continually interacting with hundreds of interconnected devices that would meld into the background and effectively become invisible to those who use them. In his paper [167] Weiser wrote that:

> "The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it." [167]

The statement illustrates Weiser's belief that for technologies to truly become an integral part of our daily lives it is necessary for them to disappear from our awareness, to be so ingrained in our psyche that it is not necessary to have to think about using them. This would enable computer users to focus beyond the tools and so concentrate on their goals. Ubiquitous computing defines a technology that does not require our constant attention but at the same time is ready to be used at a glace.

In developing his vision of ubiquitous computing Weiser recognised that the millions of personal computers inhabiting society were largely isolated and disconnected from it and were, in fact, mostly getting in the way of people. He recognised that computers were not disappearing into the background but were increasingly demanding the attention of users. Weiser realised that the relationship between humans and computers had to change to one that ensured computers became "vastly better at getting out of the way" [168] of people. He realised for this to happen computers needed to become part of the "natural human environment" [167] and that it was necessary for them to "vanish into the background" [167] of our society. He argues that for computers to truly become an integral part of our lives it necessary for them to totally disappear from our awareness.

> "Such a disappearance is a fundamental consequence not of technology, but of human psychology. Whenever people learn something sufficiently well, they cease to be aware of it." [167]

In considering the statement above it is clear that ubiquitous computing is more than just the development and deployment of technology into physical spaces it also encompasses how people perceive and interact with such technology. Whether or not the technology is allowed

to be "interwoven into the fabric of society until it becomes indistinguishable from it" [167] depends on how it is perceived and used. It is only when it is accepted that such technology can be thought as ubiquitous.

Ubiquitous computing requires a fundamental change in how computers are perceived and used in our society and demands a total rethink of how such systems are designed and implemented. For researchers the main question still remains: "how do technologies disappear into the background?" [167]. The answer lies in the widespread proliferation of devices that are sufficiently small and cheap enough to be built into everyday objects. The goal is to make these devices so pervasively distributed across society that they become such an intrinsic part of peoples lives that they no longer recognise them as computers. It is also necessary to have efficient, low-powered devices that can operate for considerable periods of time without having to change the source of power or maintain them in anyway. Devices need to be forgotten about and having to constantly support them in this manner does not allow them to meld into the background of society. With the increased numbers of devices expected to occupy ubiquitous computing environments it would become problematic to maintain devices in the same manner as we do with traditional forms of computing. As can be observed in the statement below, Weiser also noted that the real power of ubiquitous computing is only achieved when all the devices in the environment can interact with each other. It is therefore essential to provide a network that facilities the interconnection of devices.

> "The real power of the concept comes not from any one of these devices; it emerges from the interaction of all of them." [167]

Furthermore, for devices to become interwoven into the fabric of society it is necessary for them to understand the "human environment" [167]. If they are unable to comprehend what is occurring in the environment it makes it extremely difficult for devices to behave in a manner that will allow them to move into the background. Knowledge of the surrounding environment is an important facet of ubiquitous computing and can be consider as a prerequisite for most scenarios. The use of *context information* [42, 138, 18] has allowed many systems, some of which are discussed later in the chapter, to cope with this requirement. They use context in a number of ways to adapt the system to meet the requirements of the environment. For

instance, to reconfigure a system to handle the changing availability of resources, or to adapt the behavior of the system to meet the requirements of the user.

Weiser observed [167] that the technology required for ubiquitous computing comes in three parts: firstly, cheap, low-powered computers; secondly, software for ubiquitous computing applications; and lastly, a network that brings them all together. The thesis concentrates on the second requirement with the aim of providing a highly decentralized method for organising components of a ubiquitous computing environment that supports spontaneous interaction between entities and provides robust system-wide behavior. The objective is to develop a framework capable of supporting Weiser's vision for ubiquitous computing. The next section investigates the challenges in building such systems, while the following section reviews the efforts made be previous projects in tackling this objective.

2.2 Building System Software for Pervasive Computing

Weiser's vision of pervasive computing brings with it challenges not normally associated with traditional forms of computing: devices embedded in physical objects and places; the often restricted computational and network resources found in these types of environments; the high degree of mobility that is inevitable with users carrying objects from one location to another; the often inaccurate readings obtained from sensors; and the use of wireless networks to interconnect devices in the environment. Together with Weiser's vision of having technology meld into the background of society, they make supporting pervasive computing a complex and difficult task. A number of research efforts [86, 135, 81, 50, 92, 40, 126] have identified similar challenges in developing system software for supporting pervasive computing and in the discussion to follow the thesis focuses on those challenges that appear to be common to all.

2.2.1 Integration into the Real World

Kindberg et al. [86] note the physical integration of computers into the world as one of the main characteristics and challenges of pervasive computing. They argue that this can only

be achieved by providing low-level software interfaces to physical sensors and actuators, and high-level abstractions that allow applications to sense and interact with the environment.

It is therefore essential for the system software used to support these applications to abstract the complexities of dealing with the real world. It is not practical for applications to deal with all the low-level details of the physical world. It is necessary to provide high-level abstractions that applications can use to understand their environment and interact with it. The challenge for system software is to deliver appropriate levels of abstraction that allow applications to have enough understanding of the real world to enable them to disappear into the background while ensuring there is sufficient low-level support for retrieving and manipulating the physical environment.

Dey's Context Toolkit [44, 43] provides one example of how this can be achieved. The toolkit consists of three main types of components: widgets, servers, and interpreters. The context widgets provide applications with a software abstraction that allows access to context information while hiding the complexities of dealing with sensors. Context servers collect context information related to particular entities and allow applications to receive notifications of the context of particular entities. Context interpreters are used to transform context information into different formats or interpretations and can be queried by applications. The design of the Context Toolkit provides low-level abstractions for dealing with sensors while also providing high-level abstractions that can be used by applications in understanding their surrounding environment.

The Geometric Model [22] used in the EasyLiving [146] project is another example. The Geometric Model provides applications with a spatial view of the entities in their surrounding environment. The model uses measurements from sensors to define geometric relationships between the entities. Once enough sensor data has been retrieved, EasyLiving applications can query the model to explore the different relationships between the entities. The Sentient Computing effort at AT&T Laboratories in Cambridge [2] provides applications with a similar model of the real world that can be shared between users and applications.

2.2.2 Adapting to a Changing Environment

Henricksen et al. [81] argue that adaptation is necessary in order to overcome the intrinsically dynamic nature of pervasive computing environments. Kindberg et al. [86] emphasise the need for adaptation in coping with the *volatility principle* where they state that the design of such systems should assume that the set of participating users, hardware, and software is highly dynamic and unpredictable. It is clear from this that the mobility of users, devices and software components leads to a situation where the virtual and physical environments are continuously changing. Moreover, the changing behavior of users makes it necessary for pervasive computing systems to have an inbuilt ability to adapt their configuration to both take advantage of available resources and to meet the changing requirements of users.

Adaption needs to occur at a number of points in a pervasive computing system to allow, for instance, the adaption of content, the tailoring of user-interfaces for different devices, and to make optimal use of the resources available in the environment. In the Intelligent Room [64] an agent-based system [35] is used that dynamically adapts the binding between agents to meet the changing resources and capabilities of the environment. The Stanford Interactive Workspace Project [78] uses a smart clipboard that allows content from one application to pasted into it and copied from it by other applications, in the process, transparently adapting the content to meet the requirements of the other application. The Gaia [134] project's application framework [69, 131], based the on the Model-View-Controller [88] user interface paradigm, allows the separation of inputs, outputs, and the processing parts of the application making it possible to dynamically build and adapt applications to suit the available resources. For example, by changing the view component it is possible to adapt the user interface to reflect the device used by users to view the application. The challenge for pervasive computing systems, as highlighted in [86], is that adaption needs to take place without human intervention.

2.2.3 Interoperability of Pervasive Computing Components

The development of pervasive computing systems, as indicated by [50] and [135], should be accomplished in an accidental manner over a period of time and not in any structured manner.

The system evolves as users introduce new components into their physical environment. Over the life time of a system it can be expected that a wide range of heterogeneous components will be integrated into the environment or will potentially move through it on a regular basis. It is not possible to "reboot the world, let alone rewrite it" [86] every time additional components need to be integrated into the system. These types of systems need to be able to grow organically and decay gracefully with the environment as new components are added or old ones upgraded or removed. It is necessary for components to be able to spontaneously interoperate to ensure their seamless integration into the environment. Kindberg et al. [86] describes this as *spontaneous interoperation*. They believe it to be one of the main characteristics of a pervasive computing system. Edwards et al. describe it as *impromptu interoperability* [50] - not just the simple ability to interconnect, but the ability to do so with little or no advance planning or implementation.

The underlying goal is to ensure that components of varying functionality and origin can spontaneously interact with each other with little or no prior knowledge. Henricksen et al. [81] suggest that to accomplish this components will need to be able to dynamically acquire knowledge of each other's interfaces and behaviors in order to learn how to interact with unknown components. Edwards et al. [50] believe that it is implausible to expect all classes of devices or services to be known to others and that further work should concentrate on standardising communication at the syntactical level (protocols and interfaces) and leave the semantics of component interaction to developers. Davies et al. [40] argue that the design of such systems should be done in an open and extensible manner so that components can be rearranged to form applications unforeseen at the time of deployment. Kindberg et al. [86] considers a number of approaches to tackling the problem of spontaneous interoperation, including service discovery systems that dynamically locate components that match the criteria of the component, and the provision of a common interoperation model that allows all components of the pervasive computing system to interact.

The challenge is to ensure the spontaneous interoperation between pervasive computing components. The goal is to minimize the role of humans in resolving the tensions of integrating these components seamlessly into the environment so that they can move into the background

of society and become invisible.

2.2.4 Scalability

One of the features of pervasive computing is the large number of devices and software components likely to be involved in any environment. It is only to be expected as these environments grow in size that the intensity of interactions will increase with the number of devices, software components, and users inhabiting the space. Thus, the scalability of pervasive computing systems is considered [135, 81, 49] a critical problem due to the often limited bandwidth, energy, and computational power of devices typically founded in these types of systems.

Satyanarayanan [135] suggests one avenue of research is to look at the idea of *localised scalability*. They recognised that the density of interactions with any particular entity falls off as one moves away from it. In other words, the interactions with nearby entities are of more relevance to a user than those occurring at a distant location. Satyanarayanan et al. argue that pervasive computing systems can use this fact to obtain scalable solutions by severely reducing the interactions between distant entities. Kindberg et al. [86] draws a similar conclusion with the *boundary principle*. He suggests that pervasive computing system designers should divide the world into physical environments with boundaries that demarcate their content. The idea is to use administrative, territorial, and cultural considerations to define each environment. The objective is not to have one single pervasive computing system that is unable to scale but numerous systems managing different parts of the world.

A number of projects have utilised these ideas to ensure scalable solutions can be obtained. For example, the Intelligent Room [64] groups agents together into societies representing different users and physical spaces. Each society can then manage its own resources and control how it interacts with other societies and users. Aura [56] draws on Kindberg et al. [86] notion of the boundary principle to scale their architecture, in that, they define the boundaries of the environment administratively and interconnect the environments through a nomadic file system [136]. For pervasive computing the challenge is be able to allow these systems scale efficiently to large numbers of devices and software components without affecting the experience of those who use them.

2.2.5 Robustness

It is accepted [86, 40, 126] that due to the highly dynamic state of pervasive computing environments that failure is to be expected. The occurrence of failure can, in part, be attributed to the increased use of wireless networks. The limited range and interference due to nearby structures are the usual causes for the unreliability of these types of communication networks. The failure of hardware due its fragile nature or the limited lifetime of batteries also leads to the routine failure associated with pervasive computing. The often inaccurate readings from sensors contributes to the unreliability of the system software. The inherent mobility of users and components also contributes to failures. System developers have to realise that failure is common place in these types of environments and to achieve robust system behavior it is necessary to address the problem from the start.

Pervasive computing systems have tackled the problem in a number of different ways. The Interactive Workspaces Project [78] uses a decoupled communication model to provide applications with asynchronous communication. The loose coupling of components ensures that applications are less affected by the failure of other components in the system. Components of the Gaia [134] infrastructure periodically advertises their presence with a heart beat. The system removes the failed components when a heart beat has not been received after a designated period of time. It then rearranges the remaining components to meet the requirements of the environment if possible.

The overall challenge for pervasive computing systems is to be able to degrade gracefully with disturbances in the physical and virtual environments without adversely affecting the experience of users in the space. To ensure that transient failures do not cause cascading failures, and that in recovery the whole system should not be made unavailable.

2.2.6 Security and Privacy

Both Davies et al. [40] and Satyanarayanan [135] note the social and legal difficulties that pervasive computing has with privacy. They recognise the concern of users and the possibilities for these systems to be abused. The issue is particularly complicated, in that, to allow devices and applications to adhere to Weiser's vision of calm computing [166] they must be able to

obtain information about users to anticipate what is required from them. However, this knowledge is sensitive and can easily be exploited against a user and potentially lead to a serious loss of privacy. The consequences are that users will become dissatisfied and unwilling to participate in the environment. In these cases users have to be able to trust that their information will not be misused or ensure the mechanisms used to obtain the information maintains the anonymity of the user.

Kindberg et al. [86] also highlights the problem of security, which is very much interrelated with the issue of privacy in many ways. Pervasive computing systems need to prevent the unauthorised use of devices to prevent their misuse and access to private data. For people the decision to grant access is made intuitively through indept understanding of the trade-offs of conceding to a request. In pervasive computing traditional methods of securing computing systems are not easily applied when the spontaneous interoperation between components is sought. This is particularly the case when components do not have priori knowledge of one another or have a trusted third-party on which they can rely on. In these situations it is necessary to determine the trustworthiness of users while maintaining the balance between granting access and preserving the privacy of users.

A number of projects are in the process of addressing these issues. For instance, the SECURE project [24] has built a trust engine for determining the trustworthiness of users and of components within an pervasive computing environment. Priyantha et al. [128] have also built a location-based system, called Cricket, that allows a user's trusted device to passively determine its location, hence, maintaining the user's anonymity within the environment.

2.2.7 Programming Frameworks

In [86], Kindberg et al. wonder what it means to write a "Hello World" for a pervasive computing environment. In doing so they recognise the fact that building applications for these types of environments is very different from developing traditional applications. Programming frameworks used in pervasive computing have to address different concerns to those found in more traditional scenarios, in that, they must focus on minimising the distractions to users, on supporting the integration of devices or applications into the environment, and on facil-

itating the incremental development of environments. It is necessary for such programming frameworks to define suitable programming abstractions that separate the complexities of dealing with underlying infrastructure - sensors, actuators, and networks - while providing developers with an expressive means of constructing applications that are capable of adapting to the environment. The design of such frameworks must also consider that pervasive computing environments are developed in an incremental fashion over a period of time [86, 50]. Frameworks have to be designed in such a way that allows them to support the incremental construction and improvement of solutions without adversely effecting the rest of the system. Henricksen et al. [81] also argue that there is a need to provide methods that allow for the rapid development and deployment of pervasive computing environments. They suggest that such tools are required to support the number and diversity of software components in such environments. The challenge for pervasive computing is to provide a programming framework with a suitable programming abstraction that allows for the rapid and extensible development of pervasive computing systems.

2.2.8 Summary

In the previous sections the chapter outlined some of the main challenges facing pervasive computing in building system software that can support the development of these types of environments. In general, the challenges are all interrelated with Weiser's central idea of making computers disappear from our awareness: pushing them into the background of society so that they can truly become an integral part of our lives.

2.3 Pervasive Computing Projects and Initiatives

A number of research initiatives and projects have been investigating different aspects of pervasive computing since Mark Weiser first outlined the area of research in the early 1990's. The following section explores a number of these projects and investigates how they address the challenges of building system software for pervasive computing environments. The projects surveyed reflect current trends in pervasive computing research while also showing the diver-

sity of approach used in overcoming these challenges.

Section 2.3.1 describes the user-centric approach taken by Aura [56] in managing the tasks of users as they move from one pervasive computing environment to another. The Ambiente project [153], described in section 2.3.2, demonstrates how a human-centered approach supported through the use of computer augmented artifacts can assist in the collaboration of users. Section 2.3.3 shows how the Adaptive House [108] uses a proactive approach in learning the behavior of users in order to anticipate their needs in the future. The Intelligent Room [64] in section 2.3.4 demonstrates how an agent based system can be used to support the activities of users in a pervasive computing environment. Section 2.3.5 outlines the sentient computing [2] approach developed by AT&T Laboratories, in particular, the section describes how a geometric model of the environment in combination with an API that exploits spatial facts within such a model can be used to change the behavior of applications. Section 2.3.6 describes the work of the Stanford Interactive Workspace Project [78]. They have developed an approach that provides users of pervasive computing environments with access to services that support their work and collaboration with other users in a workspace environment. In developing the system the Interactive Workspace Project has focused on providing a portable system that is both extensible and robust. The Gaia project [134] described in section 2.3.7 demonstrates how pervasive computing environments can provide users with access to services that are independent of the user's location. The approach is centered on managing the computational presence of mobile users ensuring that as users move from one location to another that their associated computational sessions can be migrated with them. Section 2.3.8 describes the approach taken by the One.World project [60, 61, 62] in constructing pervasive computing applications for highly dynamic environments. They have focused on dynamically connecting devices and coordinating their actions seamlessly to help users complete tasks in the environment.

The proceeding sections outline in greater detail the methodologies used in these research projects and the mechanisms applied in the integration of devices and applications into a pervasive computing environment. Other background information relating to the thesis, such as the concept of stigmergy and the scripting languages used in pervasive computing systems,

are reviewed in later chapters. The next section begins with a review of the Aura project.

2.3.1 Aura

The aim of the Aura project [56] is to provide users of pervasive computing environments with access to services that are independent of the user's location. To achieve this the Aura project has focused on managing the "computational presence" of mobile users ensuring that as users move from one location to another their associated tasks are able to move with them in a transparent manner. In so doing, Aura takes a user-centric approach to managing users' tasks and to adapting the available resources to meet the needs of users in a proactive manner.

2.3.1.1 A Task-Driven Approach

Garlan et al. [56] use a task-driven approach for Aura. Each task represents a particular aspect of what a user wants to achieve. For example, a task may be the organisation of a conference, or the preparation of a presentation by a user. The user's computational presence in the environment is modeled as a collection of these tasks. As users move from one environment to another the tasks associated with them are migrated by Aura so that the user may continue to work on the tasks. During this process Aura also renegotiates the support for the tasks in relation to the resources and capabilities of the given environment. The tasks in Aura are represented as a series of abstract services. Examples, of abstract services could be a text editing service, or a video playing service. Suppliers of these services are provided at each location at which Aura is present. When a user starts a task the system brings up all the services and files associated with that task. On finishing a task, or when a user moves out of the environment all the information associated with the task is checkpointed and stored for later retrieval.

2.3.1.2 Aura Architecture

The Aura architecture [148] is shown in figure 2.1. Coda [136] provides Aura with support for nomadic, disconnectable, and bandwidth-adaptive file access. It supports the mobility of users between environments and ensures the robust behavior of the system through its

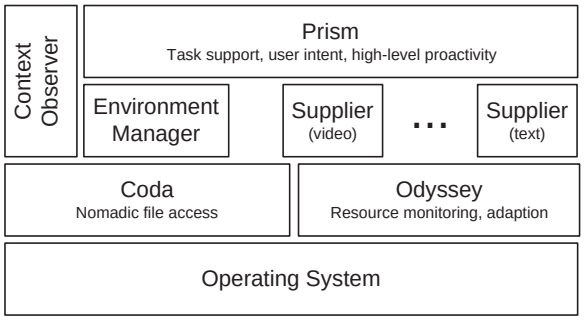


Figure 2.1: Aura architecture.

nomadic file system. Coda also provides Aura with some support for security and privacy, in that, the data being stored is encrypted. Odyssey [137] is used to monitor resources within the environment and to adapt the configuration of Aura to maximise the use of available resources in each of the environments.

Prism, the task manager in combination with the Environment Manager and Context Observer provide the functionality for supporting the task-driven approach adopted by Aura. In each environment an instance of each of these components is provided. The Environment Manager keeps track of all the suppliers of services that are available in the environment and ensures that the most appropriate services are chosen when reconstructing a task for a user. It is also responsible for providing access to the distributed file system, Coda.

Prism manages and coordinates the migration of all information related to user tasks from one environment to another. It also, when necessary, finds alternative configurations for supporting user tasks when the available resources are insufficient to meet the successful completion of the task. It achieves this by monitoring the services supporting the task. When a service can no longer complete the task a query is made to the Environment Manger by Prism for an alternative service that is capable of doing so. The abstract notion of services helps Aura to adapt the available resources to the tasks in each of the environments and in so doing provides it with a flexible approach to adapting to the resources and capabilities that are available. Prism also manages the user transition between different tasks and monitors the context observer for changes in context information that might effect the system. If any

changes are observed Prism makes the necessary adjustments for the pervasive computing environment to continue to operate.

Context information is provided by the Context Observer, the design of which, is based on [80]. It uses a database abstraction, with an SQL-like query language, to retrieve context information from the environment. It provides the Aura architecture with the ability to integrate into the environment abstracting the low-level complexities of dealing with sensors and providing a high-level abstraction that can be used to query the state of the physical environment. The information gained allows Prism to adapt its behavior to suit the users in the environment.

It would also appear that Garlan et al. [56] draws on Kindberg et al.'s [86] notion of the boundary principle to scale their architecture. They define the boundaries of an environment administratively. In practice this appears to correspond to a physical location such as a room or building. In each of these environments their exists a single instance of the architecture as described above.

2.3.1.3 Developing Pervasive Computing Environments with Aura

Developing a pervasive computing environment with Aura is centered on the provision of service suppliers for the different types of services and various resource configurations in the different environments. In the current implementation of Aura the majority of service suppliers wrap existing applications. For instance, the text editing service is based on Emacs and Microsoft Word, while the video playing service is provided by Media Player and Xanim.

The Aura architecture makes the assumption that the different types of services it requires to reconstruct a task will be available in each of the environments to which the user moves. While this may be the case for the majority of common services it may not be for all. In these cases Aura may not be able to start a task for the user at that location. While it would appear that it is relatively easy to install a supplier for a service at a certain location it may not be possible to use that type of service until it has been rolled out to a sufficient portion of the Aura environments that the user inhabits as it would not be possible to reconstruct those tasks based on that service.

2.3.1.4 Summary

Aura provides a user-centric approach to the development of pervasive computing environments. It pays particular attention to the migration of a user's computational presence from one environment to another. It models this presence as a series of tasks that are represented as a collection of abstract services and files. As users move into an environment their tasks move with them and are reconfigured to map onto the resources that are available in the environment.

2.3.2 Ambiente's Cooperative Project

Streitz et al. developed the concept of Cooperative Buildings [153] to provide flexible and dynamic environments that provide cooperative workspaces supporting communication and collaboration between users. Their aim was not only to support human-computer interactions but to assist in human-human cooperation and communication by using real world artifacts as the interface to information and as the means of supporting the collaboration and communication between users. iLand [154] is the realisation of this concept and provides a workspace for assisting human creativity within a meeting scenario. It uses computer-augmented objects - Roomware [153] - to provide information and support for group and individual interaction within the iLand environment. In taking this approach Streitz et al. have used a human-centered design where the human may also be part of a group or organisation.

2.3.2.1 Roomware

The concept of Roomware [153] was conceived by Streitz et al. as a way of creating flexible, dynamic landscapes, that could integrate real architectural spaces with virtual information spaces in a way that supports the cooperation and communication of one human to another. So far Streitz et al. have created a number of Roomware components; an interactive electronic wall (*DynaWall*), an interactive electronic table (*InteracTable*), networked chairs (*CommChairs*), and a smaller interactive table called the *ConnecTable*. All of which can be seen in Figure 2.2.

DynaWall is a large interactive display of 4.5 m in width and 1.1 m in height. It is

Figure 2.2: Roomware: ConnecTable, CommChair, DynaWall, InteracTable.

constructed from three touch sensitive rear-projected whiteboards. The display is used in iLand to aid teams to display and interact with large amounts of information. Users interact with the display by using gestures to create, move, and delete information objects from the display. The CommChairs are mobile chairs with either an in-built computer or docking point for a laptop. The chair allows users to communicate and share information with people in other chairs or with other Roomware components such as DynaWall. From the chair the user can annotate these remote workspaces as well as making their own personal notes within their own private workspace. The InteracTable is a table with a vertical touch-sensitive display embedded into the top of the table. It is designed to be used by groups of up to six people. Gestures are used by the group to manipulate the different information objects and annotations can be provided through the use of voice and/or pen. ConnecTable is a smaller version of the InteracTable and has been designed by Streitz et al. for individual work or for cooperation between small groups. Each of these components, with the exception of DynaWall, use wireless communication and have their own independent power source. This allows the users to configure the Roomware components in the desired way. However, the movement is limited to the range of the wireless communication as iLand is based on a client/server architecture.

2.3.2.2 iLand Infrastructure

Streitz et al. use the Roomware components to support the collaboration and communication between users in the iLand environment. In order to support the functionality of these components Tandler has developed the Basic Environment for Active Collaboration with Hypermedia (BEACH) [156, 157] system for supporting synchronous collaboration and interaction between Roomware components. BEACH is horizontally organised in four layers - core, models, generic, modules - which define increasing levels of abstraction and vertically by five models - interaction model, physical model, user-interface model, tool model, and document model - that separate the basic concerns of the infrastructure. The abstractions provided by BEACH, in particular the physical model, help to remove the low-level complexities of dealing with the real world and so aid the physical integration of the Roomware components into the iLand environment.

To ensure a clear separation of concerns Tandler uses different models for interaction, the physical environment, user interface, tools, and documents. The document model describes all objects related to a document. The tool model provides descriptions of tools that can be used in conjunction with the user interface, for instance, toolbars and document browsers. The user interface model defines alternative user interfaces that can be used with each device in the iLand environment. The physical model provides representations of the real world. The interaction model defines the means by which users can control and manipulate the different components of the system. All the models are implemented as shared objects except for the interaction model object which is local to each Roomware component. The sharing of objects allows for several users or devices to access the objects at the same time.

The core layer of BEACH, which is based on COAST [145], provides the means of achieving this. COAST provides BEACH with a shared-object space that allows the distribution, replication, and synchronisation of objects across the iLand environment. The core layer also provides additional functionality for event handling and sensor management. The layer ensures the spontaneous interoperability between the Roomware components in the iLand environment. However, COAST is based on a client/server architecture where each of the Roomware components runs one or more BEACH clients that synchronise with a central

server. Such an architecture may be subject to issues of scalability and reliability due to the use of centralised components to coordinate the Roomware components.

Directly above the core layer is the model layer. The model layer provides an implementation of the basic models that are used in implementing the higher layers. For instance, the layer provides interfaces for manipulating documents, for user interfaces, and models for different styles of interaction within the iLand environment. Based on the model layer the generic layer provides a set of components that support the basic functionality that is required for supporting most teamwork and meeting situations. For example, the components at this level support informal handwritten scribbles, as well as private and public workspaces for collaboration between users. In additional to the generic layer is the modules layer, which provides extra support for defining functionality for specific tasks not handled by the components in the generic layer. Currently, this layer is used in iLand to support creative teams in collecting ideas during brain storming sessions.

It should be noted that BEACH requires a considerable amount of computational resources and is not suitable for small devices such as PDAs. Also, the infrastructure requires continuous network connectively to support the synchronous collaboration between devices in the iLand environment. This does not lead to a robust pervasive computing system as wireless communication used in the roomware components is not considered to be reliable. Also, the infrastructure is primarily built for supporting meetings, braining-storming sessions and it would appear that it may be difficult to use it develop other types of environments.

2.3.2.3 Developing Roomware Components with BEACH

Developing pervasive computing environments with BEACH is centered around the creation of Roomware components for the environment. In the iLand environment a number of such components - DynaWall, InterTable, CommChairs, and ConnecTable - were developed to assist groups of people in creative sessions. In BEACH the development of these Roomware components is centered on the generic layer as this layer define the concrete classes that are used to implement the generic support for the components. For example, the DynWall consists of three machines each with a wall mounted display. The physical model at this layer would

define a roomware component of three machines with their displays combined into one large display area. The user interface model would define the display as one large segment in which a document browser is placed. The document model would define the workspace.

2.3.2.4 Summary

Ambiente's Cooperative Project have developed a system for supporting the communication and collaboration between groups of users in a workspace environment. Their approach is centered around the concept of Cooperative Buildings [153] and the use of computer-augmented objects to provide information and support for group and individual interaction in a pervasive computing environment. The iLand [154] environment is the initial prototype of the concept, which is based on a collection of Roomware components that are supported by BEACH. While BEACH is effective in abstracting the complexities of supporting the collaboration of users there are concerns with its ability to scale to larger a environment and to provide a robust service in the face disturbances within the environment. The work presented in this thesis looke to address these issues through the use of techniques based on the natural phenomena of stigmergy.

2.3.3 The Adaptive House

The aim of the Adaptive House [108] is to provide a proactive approach to learning the behavior of users in order to allow a system predict the future needs of users in a pervasive computing environment. To achieve this Mozer et al. renovated a school (see figure 2.3) into a two bedroom house. In the process they incorporated a large network of sensors and actuators to sense and control various environmental aspects of the house including the lighting, heating, and ventilation. Their objective in installing the network was to incorporate the ACHE system [107] which would allow the house anticipate the needs of the inhabitants and to conserve energy within it.

To determine the usage being made of the house Mozer et al. equipped the house with a wide range of sensors to detect the state of the lights, speed of the air conditioning fans, and the temperature of each of the rooms. It was also necessary for them to have information on

Figure 2.3: The Adaptive House 1926.

the temperature of the water heater, the energy used by the heater, the outside temperature, and the energy being used by the furnace. To detect the location of the inhabitants the house was augmented with a number of motion detectors in each of the rooms. In total Mozer et al. used approximately 75 sensors to gauge the environmental state of the house. They also used 22 lighting banks to allow the ACHE system to control the intensity of the lighting as well as to turn it on and off and a number of other actuators to control the speed of the ceiling fans, to turn on and off the water heater, gas furnace and electric space heaters. All the devices were connected together over the house's power line using an X-10 network.

2.3.3.1 ACHE

ACHE [107] is the underlying system that regulates the behavior of the Adaptive House. It monitors the activities of the inhabitants to anticipate their needs and to alter the state lighting or heating in the house. To achieve this the ACHE system uses the large sensor network embedded in the house to sense the state of the house and the activity of the occupants. Mozer et al. use neural networks in combination with reinforcement learning techniques to decipher the sensor data and to predict the future behavior and usage of the house. Having determined the probable usage the ACHE system is able to adapt the different environmental systems to meet the predicted needs of the occupants and to conserve energy within the house. The basic system architecture [107] of ACHE, as shown in figure 2.4, is replicated in each of the control domains: lighting, heating, and ventilation.

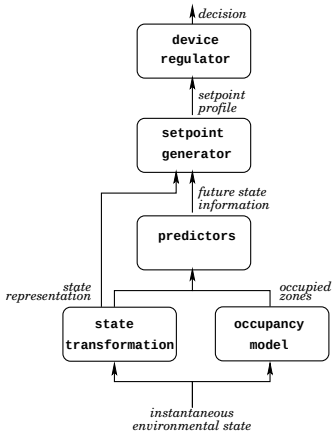


Figure 2.4: System architecture of ACHE.

The sensor readings gained from the network are fed into a state transformation component and occupancy model. These determine the current state of the house and the whereabouts of the inhabitants. The predictor components use the information provided by the state transformation component and occupancy model to predict future usages of the house. For example, the occupancy patterns of the house for the next hour or the expected water usage in that time. Using the current and future state of the environment the setpoint generator determines the most appropriate profile to which to set the different environmental systems. The profile is passed to the device regulator to modify the devices in the house to achieve the required state.

ACHE has been shown [110, 109] to provide quite an effective mechanism for adapting the environmental aspects of a residential home to the needs of the inhabitants. However, as the system runs on a single server it is unclear how it could scale to a larger building or whether it be used across several buildings. There is also a concern as to how robust the system is to failure of the main server or whether the data stored in the occupancy models, state transformation components, or predictor components can persist across reboots. It should also be noted the Adaptive House is static in that there is no expectations that new components will be incorporated into the environment. This questions the ACHE system's ability to spontaneously interoperate with new components.

2.3.3.2 Summary

The Adaptive House shows how it is possible to adapt a system over time to the changing behavior of users through the use of neural networks and reinforcement learning techniques. While the ACHE system has proven to be quite effective in achieving this it neglects in other respects to fully support pervasive computing. For instance, how the system is to scale to large environments, or how the spontaneous interoperation between arbitrary components is achieved. It should also be noted Mozer et al. do not describe a programming model that can be used to develop other types of pervasive computing environments and as such would appear to be limited to the control of environmental systems within a residential house. The work presented in the thesis addresses these issues through the provision of an alternative programming abstraction for developing pervasive computing environments.

2.3.4 Project Oxygen - The Intelligent Room

The Intelligent Room [34], developed by Coen, was the initial project that started MIT's research into pervasive computing. MIT's interest has now evolved into an initiative called Project Oxygen that includes a new version of the Intelligent Room [64]. The Intelligent Room is a conference room containing a number of cameras, microphones, sound system, smart boards, visual displays, and a number of other sensors and actuators. The aim of the project is to provide a human-centered approach capable of supporting the activities of users in a pervasive computing environment. The underlying system for the environment is based on an agent system called Metaglue [121, 35].

2.3.4.1 An Agent-Based System

Hanssens et al. [64] use an agent-based system to control and develop applications for the Intelligent Room. In their approach an agent represents any software component with the ability to communicate and to provide functionality to other agents in the environment. These may include an agent to control a light switch or an agent to provide a fully featured word processing application for users. Hanssens et al. group these agents in societies to represent and to act on behalf of a user or a particular space. By structuring the agents in this way they

are able to scope the interactions within the societies and to limit unnecessary interactions between societies making it possible for the system to scale in an acceptable manner.

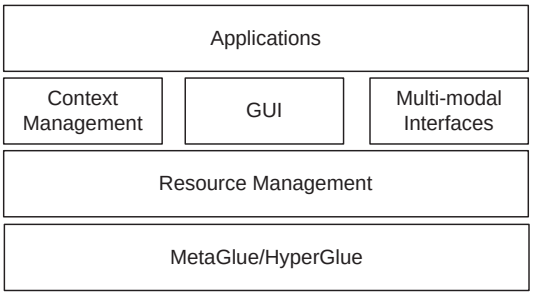


Figure 2.5: Intelligent Room Architecture.

Agents are used as the basic component for the Intelligent Room. The system architecture (see figure 2.5) consists of four layers working at increasing levels of abstraction. The communication layer - MetaGlue [35] and HyperGlue [64] - contains software components that are used to mediate communication within a society and also between societies. The next layer - Rascal [54] - coordinates access to the resources of the society. The next layer consists of a context-aware component - ReBa [90] - that uses context information to provide relevant services to users, a GUI management component that controls the graphical user interfaces for applications, and a multi-modal component that contains methods for different modes of user input, such as gestures and speech recognition. At the top is the application layer which interacts directly with users.

Metaglue [35] is the agent system used by the Intelligent Room. It is developed in Java to provide low-level support for building software agents for pervasive computing environments. It is designed to aid communication and provide fault tolerant mechanisms for coordinating the actions of agents. The system uses a post-compiler that runs over the Java class files to generate new byte code that runs on the MetaGlue Virtual Machine. To support the development of agents Metaglue adds a number of language primitives to the Java language, a detailed description of which can be found in [121]. They are used by MetaGlue to dynamic load and connect agents together, and to tie agents to particular resources within the envi-

ronment. Communication between agents is achieved via a publish-subscribe mechanism or through the direct invocation of methods. The MetaGlue system is also designed to handle the failure of agents, in that, it is capable of restarting crashed agents in a manner that is transparent to other agents. Using MetaGlue in the Intelligent Room has allowed Hanssens et al. to provide a robust method of handling the failure of components and a means of abstracting such issues from higher-level agents. The dynamic binding of agents also provides the Intelligent Room with the means of adapting to changing agent configurations.

HyperGlue [64] enables inter-communication between societies of agents. In each society there is an *ambassador* agent that represents the grouping to the rest of the environment. These ambassador agents advertise their society and locate other societies so requests for resources and exchange of information can be made. The discovery mechanism used by ambassador agents is based on the Intentional Naming System (INS) [3]. The common communication model that MetaGlue provides in combination with the discovery mechanisms provided by HyperGlue ensures that the agents in the different societies are able to spontaneously interoperate with each other. Such functionality allows Hanssens et al. to seamlessly integrate components into a pervasive computing environment.

Rascal [54] is used to coordinate how the resources in the pervasive computing environment are used. The main function of Rascal is to arbitrate between requests for resources and to enforce access control for the system. It is composed of a knowledge base, a constraint satisfaction engine, and a framework that allows interaction with agents. The knowledge base is used to hold information about resource requirements, while the constraint satisfaction engine arbitrates the requests of agents for similar resources. Each society of agents manages and controls access to their own resources. Cross-space requests for resources are made through the ambassador agent. The use of Rascal provides the Intelligent Room with a way of managing resources and a means of adapting the configuration of agents to meet the availability of resources in the environment.

Controlling the behavior of each society is a behavioral system called RaBa [64]. The design [90] of which is heavy inspired by the work of Brooks [15] on subsumption and in part by that of Williams et al. [172]. Subsumption is a layered architecture that was developed

by Brooks to control the behavior of mobile robots when moving through unfamiliar environments. The system works by exploiting various levels of competence. Each level defines a particular class of behavior for the robot. Higher levels define more specific desired classes of behaviors. In addition, these higher levels can suppress, or subsume, lower-level behaviors by changing their output. Hanssens et al. view the Intelligent Room as an immobile robot and have use Brooks's layered approach to control the behavior of their pervasive computing environment.

ReBa collects context information from agents to build up a higher level representation of the activity being performed by users in the physical environment. Hanssens et al. call this representation the *activity context.* For each activity (or sub-activity) being performed by a user a *Behavior agent* is used to represent it. In its simplest form each behavior contains a rule that responds to a user action by performing a particular reaction (output). Depending on the order of activities performed by users a series of behaviors can be activated by the system. An activity can be performed within another activity creating a layering of behaviors activated one on top of each other. For example, a presentation behavior can be activated on top of a meeting behavior, or it could be activated on top of a casual gathering behavior creating a different reaction in the physical environment. The ReBa component provides the Intelligent Room with the means of changing its behavior, in so doing, it allows the Intelligent Room to adapt to meet the needs of users.

User input and graphical user interfaces are managed separately from application logic. An application can receive user input via speech, gesture, and pointing devices. To allow users in different locations to interact with the same application Hanssens et al. have created a distinction between the application agent and the graphical user interface of the application. To transparently manage each user interface a GUI Manager agent runs locally on each device with a display. The manager can request a user interface from the application and load it locally onto the screen where a user can control it and share it with other users. Separating the inputs and outputs from the application logic makes it possible to dynamically compose applications to suit the available resources within the Intelligent Room. This makes it easier to adapt the applications to environment and to the needs of the user.

2.3.4.2 Using Agents to Build Pervasive Computing Applications

In the approach advocated by Hanssens et al. the development of pervasive computing environments is centered around developing agents for the various sensors, actuators, computational devices, and application logic used in the environment to represent the different physical spaces or users. The agent system used by Hanssens et al. and the abstractions that it provides helps to remove the complexities of dealing with the physical environment, easing the physical integration of devices while also aiding the development of applications for these environments. So far the system has been been used to build the Intelligent Room [64] and a number of application for it. For instance, Fire [55] is an information retrieval interface that allows users to use more natural modes of communication such as speech and gesture to retrieve information from the Internet. The Meeting Manager [120, 113] provides support for users who are having a meeting within the Intelligent Room. It helps users plan and manage a meeting while at the same time records the meeting with contextual cues that can be used to search the recording of a meeting at a later stage. It also provides a summary of the meeting that it sends to each participant at the meeting.

2.3.4.3 Summary

The Intelligent Room provides a human-centered approach to the development of pervasive computing environments. They pay particular attention to supporting the interactions between users and also the interaction of users with the physical spaces they occupy. To support these interaction they have developed an architecture based on the agent system MetaGlue. The agents are grouped into societies to represent and to act on behalf of various users and physical spaces.

2.3.5 AT&T Laboratories - Sentient Computing

Sentient computing [2] uses sensors and other resource data to maintain a model of the real world that can be shared between users and applications. Addlesee et al. use this model to allow applications to make observations of the real world and to adapt their behavior correspondingly. The system developed to harness this approach provides a geometric view

of the world, tracking the location and monitoring the state of physical objects and users with it. The developement of applications is achieved through an API that uses spatial facts derived from the model to trigger changes in application behavior.

2.3.5.1 A Sentient Computing System

The sentient computing system uses a three-tier architecture [2, 65] to maintain a model of the real world. On the top tier are resource monitors and location sensors [165] that feed the system with information on the state of the physical objects. The applications that use the model are also located at this level. In the middle tier are a set of CORBA objects that represent the physical objects within the environment. At this level is also a spatial monitor that determines the spatial relationships between physical objects. On the bottom tier is an Oracle database that is used to provide persistence for the system.

An object-oriented data modelling language - Ouija [149] - is used to generate the CORBA objects on top of the database. These objects are stored in the database as rows of data and associated operations are written in PL/SQL. PL/SQL is a procedural extension to SQL that was developed by Oracle to allow manipulation of their database. The Ouija modelling language provides a CORBA mapping for PL/SQL and a means of persistently storing the state of objects. This allows the sentient computing system to handle failure and to provide a more robust service to applications. The CORBA objects generated with Ouija are used to represent the various physical objects within the environment. The resource monitors and the location service update the state of these objects via the corresponding CORBA objects. Changes in the state of the physical objects, such as location, are then propagated to applications via an event service.

Resource monitors are installed on all networked machines and are use to provide information on machine activity, resource usage, and network point-to-point bandwidth and latency. They periodically report changes to objects in the database via the CORBA interface generated by Ouija. The location system - BAT [165] - provides the primary source of information on the location of physical objects. It is an extremely accurate location system though it requires the preinstallation of base stations into areas where the location of objects is required.

All location information is also feed back into the system.

Conceptually, the architecture provides a centralised system that maintains a model of the real world through the use of sensors and which allows applications to adapt their behavior through observations of this model. While the use of a centralised system may ease the implementation there has to be concerns that the chosen architecture will not be able to scale to incorporate larger areas or more detailed models of the real world. However, the high-level abstraction that this approach provides does aid the physical integration of applications into an pervasive computing environment while also allowing them to adapt their behavior to changes in the environment. Though, it should be noted that there is no common coordination mechanism that allows for interoperability between arbitrary application components. Applications can only use the observations from the model to change their behaviour and have no means in this architecture of communicating with other components to coordinate their behavior. There is also no means of securing access to those viewing the model which may lead to concerns of privacy for users.

2.3.5.2 Programming with Space

To develop applications Addlesee et al. use an API that exploits spatial facts between physical objects to trigger changes in application behavior. This is achieved through the use of the spatial monitor located on the middle tier of the architecture. The spatial monitor translates absolute location events generated by the system into relative location events. These events can be used by applications to determine relative spatial facts about physical objects of interest to them, for instance, whether a person is standing in front of a workstation or not. To receive these events applications need to associate a particular containment space around an object and to register a callback to the monitor the space for different spatial facts. The spatial monitor can then deliver relevant events to the application which, in turn, allows the application to change its behavior.

Addlesee et al. have used the approach to develop a series of follow-me type applications. For instance, using Virtual Network Computing [51] one of the follow-me applications created a desktop that followed its owner from desktop to desktop. Another follow-me application allows

a person to route telephone calls to the phone beside them. A number of other applications have also been developed such as Lifestreams [52, 2], which keeps track of the stream of documents accessed in a time-line that functions as a users diary of their electronic life, or a browser that allows people to see what is happening in the building. The majority of the applications developed can be classified as location aware systems. The use of other types of information does not appear to influence the behavior of the system in any significant way. This would seem to be a drawback as it limits what can be achieved if other forms of information could be used to change the behavior of applications.

2.3.5.3 Summary

Addlesee et al. [2] have developed an approach based on the concept of sentient computing for developing pervasive computing environments. In it sensors and other resource data are used to maintain a model of the real world that can be shared between users and applications. It pays particular attention to the geometric relationships between physical objects and uses a spatially aware API to define the behavior of applications. While the programming model provides an initiative approach to developing pervasive computing applications the underlying system is limited due to its centralised nature and particular focus on the use of location information.

2.3.6 Stanford Interactive Workspace Project

The aim of the Interactive Workspace Project [78] is to provide users of pervasive computing environments with access to services that support their work and collaboration with other users in a workspace environment. To achieve this the Interactive Workspace Project has focused on the ability to move data transparently between applications and displays within the environment, the ability to move control of applications between devices to minimise the disruption to users in collaborative sessions, and the ability of applications to dynamically co-ordinate their actions to support the activities of users. In doing so, the Interactive Workspace Project have taken a user-centric approach to mananging the interactions of users with the environment.

2.3.6.1 iROS a Pervasive Computing System

To support the collaborative sessions of users Johanson et al. have developed a system, called iROS [78], to manage the interactions of users within these types of pervasive computing environments. The component architecture of iROS can be seen in figure 2.6. The Event Heap [77] is a distributed event service based on a tuplespace model [28]. It provides the communication infrastructure for the dynamic coordination of applications. The Data Heap supports the movement of data from one application or device to another through the provision a repository that automatically converts the format of the data to best suit the application or device accessing the data. The iCrafter system [127] is used for service advertisement and invocation and is, in many respects, quite similar to Jini [162] except that invocations are mediated through the Event Heap. iCrafter also provides an additional service for the automatic generation of user interfaces for services such as a light, projector, or application within the environment.

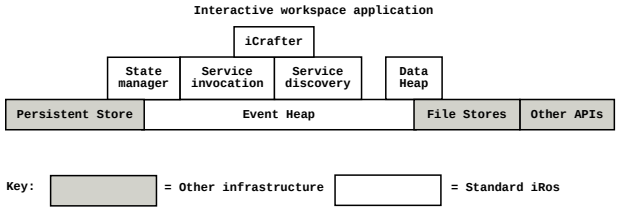


Figure 2.6: The iROS component structure.

These three subsystems - event heap, data heap, and iCrafter - are designed to handle the user modalities of moving data, moving control, and the dynamic coordination of applications within the environment. Together, they also provide additional functionality for supporting the development of pervasive computing environments. The common coordination mechanism used by the Event Heap allows for the interoperability between arbitrary components, while the loose coupling provided by a tuplespace model ensures that failure of components does not significant effect the rest of the system. The simple functionality of the Event Heap also ensures the portability of the system to a wide variety of platforms easing the inclusion of new applications and devices. However, the current implementation of the Event Heap uses

T-Spaces from IBM [174], which is a client-server based implementation of a tuple space. It would appear [77] that with this implementation there are issues with the performance and scalability of the system. Even though Johanson et al. [78] look to use Kindberg et al's [86] notion of the boundary principle to administratively bound the size of the environment to allow their system to scale there would still seen to be a problem.

The Data Heap's ability to act as a smart clipboard significantly helps the iROS system to transparently adapt content to the requirements of applications and devices. The ability to adapt is also helped by iCrafter's capability to generate user interfaces for services. This is achieved through templates which can be used to adapt the interface to suit the available resources in the environment. The iCrafter system also allows the composition of services enabling the iROS system to adapt applications to suit user requirements. This is facilitated through the use of a service discovery mechanism that is mediated via the Event Heap. The mechanism allows components of the iROS system to discover and to spontaneously interoperate with other components aiding their composition.

However, while the abstractions provided by iROS help to construct pervasive computing environments there appears to be no high-level abstraction for sensors that can be used to tailor the behavior of applications. This hinders the physical integration of applications and devices into the environment and restricts the proactive nature of such environments. It should also be noted that the iROS system is logically centralised with the Event Heap acting as the hub for interactions within the environment. While this has it advantages, as discussed above, it does act as a central point of failure for the system which questions its robustness even though Shankar et al. [126] and Johanson et al. [77] suggest that it is not significant problem. Also, the fact that applications use the Event Heap as a broadcast medium allows users to over see the communications of applications other than their own queries the ability of the iROS system to protect the privacy of users and the security of user data.

2.3.6.2 iRoom a Pervasive Computing Environment

With the iROS system the Interactive Workspace Project have created an environment called iRoom. iRoom is a meeting room that allows users to collaboratively work together. The

room provides users with wall-mounted displays, called interactive murals, and a table with a display inserted into it. These pieces of equipment provide the focus for the meeting room and for the interactions with the system. Users can also bring their laptops and PDAs into the iRoom and have them seamlessly integrate into the environment. Participants making a presentation at the meeting can use their laptop to display their material on the interactive mural. The other participants at the meeting can make suggestions or modify the material using the PointRight system [79] that allows users to control the pointer on the interactive mural from their own laptop. It is also possible for users to control other devices in the room via their laptop or PDA using a user interface generator.

2.3.6.3 Developing Applications with iROS

In the iROS system the Event Heap is the only component an application is required to use to become part of the system. However, the development of applications is more centered around the provision of services using the iCrafter subsystem. In iRoom services are used to control the lights, the projectors, and also to build applications for the meeting room. Many of these applications are existing applications that have been wrapped using the iCrafter service API to enable there inclusion within the environment. For example, Microsoft's Power Point application is wrapped in a iCrafter service to allow it to be used in the iRoom to make presentations.

2.3.6.4 Summary

The Interactive Workspace Project [78] have developed a pervasive computing system that supports the collaborative work of users within a workspace environment. It pays particular attention to the ability of users to move data between applications, for users to move control of applications between devices, and for applications to be able to coordinate their behavior in a dynamic manner. In tackling these concerns the Interactive Workspace Project built iROS a logically centralised system that manages and coordinates the interactions of users within a localised environment. An alternative system is presented in the thesis were a decentralised approach is developed to allow the ad-hoc assemble of pervasive computing environments.

2.3.7 Gaia

Similar to the Aura project [56] the objective of the Gaia project [134] is to provide users of pervasive computing environments with access to services that are independent of the user's location. To achieve this the Gaia project has concentrated on managing the computational presence of mobile users ensuring that, as users move from one location to another, their associated computational sessions can be migrated with them in a transparent manner. In so doing, Gaia takes a user-centric approach to managing the interactions of users with the environment and to adapting the available resources to the needs of users in a proactive manner.

2.3.7.1 Active Spaces

To support the mobility of users in a pervasive computing environment Roman et al. [134] define an active space as a physical space that is constrained by well-define physical boundaries containing tangible objects, heterogeneous network devices, and users performing a range of activities that are supported by a context-aware infrastructure. The aim for Roman et al. is to enhance the ability of mobile users to interact with and configure their physical and virtual environments in a seamless manner. To achieve this they use the idea of sessions. Session associate data and applications with a user. A user's computational presence in the environment is modeled as a collection of these sessions which Roman et al. call the user's virtual space. As users move from one active space to another the sessions associated with them are migrated by Gaia to allow the user to continue their work. During the migration Gaia renegotiates the support for the sessions and maps the applications and data to the available resource in the active space.

2.3.7.2 The Gaia Operating System

The Gaia Operating System (Gaia OS) manages the resources and provides the services for an active space. The design of the Gaia OS is based on a traditional operating system but at a different level of abstraction. Gaia OS abstracts a physical space and the resources it contains into a single programmable entity that can be used to build active spaces. It

looks to match the services found in traditional operating systems - program execution, I/O operations, file-system manipulation, communications, and error detection - to manage the resources and provide support for user interaction and mobility within a pervasive computing environment. It is a component-based distributed operating system that operates on top of existing operating systems and is based on the K2 operating system [87]. The system, as shown in figure 2.7, consists of a component management core around which a set of core services - event manager, context service, presence service, space repository, and context file system - provide support for users and application development.

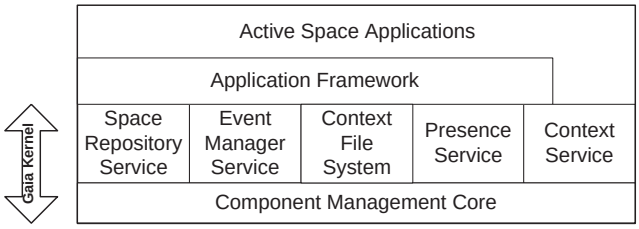


Figure 2.7: Gaia architecture.

The component management core is a component-based system that supports the distribution, remote execution, and management of components. It is responsible for creating, destroying, and uploading components to any of the computational nodes within the active space. The component management core is implemented using CORBA. The event manager distributes events and provides a decoupled communication model for the active space. The use of such a communication provides increased system reliability and ensures a robust system capable of handling failure of components. The events are used in active spaces to notify interested components of new services, the entry of people, errors, and other application specific events.

The context service [130] provides Gaia with context information that can be used to adapt the behavior of applications. The service is structured in a similar fashion to Dey's Context Toolkit [44, 43] but also uses a first-order logic and boolean algebra to frame rules and to define queries. The component helps to abstract the low-level complexity of dealing with sensors and provides a high-level abstraction that can be used by components to query

the state of the physical environment. The approach taken by the Gaia project supports the physical integration of components into a pervasive computing environment.

The presence service is used to monitor the state of resources within the environment. It keeps track of applications, services, devices, and users entering and leaving the environment. Interested parties are notified of changes through the event service. The space repository stores information on all software and hardware entities within the active space and allows applications to browse and retrieve entities. The space repository learns of new entities through notifications from the presence service. On a user entering the active space the space repository retrieves the user's profile and stores it for later retrieval. The space repository acts as a service discovery mechanism for Gaia allowing applications to locate suitable resources during instantiation. Such a services aids the spontaneous interoperation of arbitrary components in the active space and also helps with adapting applications to the available resources in the environment.

The context file system [67, 68] is a context-aware file system that uses context information to simplify access to data. The file system uses the context associated with each file to construct a virtual directory hierarchy that can be queried at a later stage by applications. In the active space it is used to store personal data and to save the state of user sessions so they may be retrieved at a later stage. On a user entering an active space the context file system automatically locates the user's data and makes it accessible to the components of the active space. It is at this stage that any active sessions are mapped to the active space. The context file system also eases access to data by reorganising and retrieving the data in a format suitable to the user's preferences or device characteristics. The use of such a file system helps to adapt the content to suit the available resources in the environment.

It would also appear that Roman et al. [134, 132] draws on Kindberg et al's [86] notion of the boundary principle to scale their infrastructure. They define the boundaries of an active space administratively. This would appear, in practice, to correspond to physical locations such as an office, or meeting room. In each of these active spaces exists a single instance of the Gaia OS that manages the resources in this space and controls how the environment interacts with mobile users.

2.3.7.3 Developing Applications with Gaia

Developing pervasive computing environments with Gaia is centered around the application framework that is implemented on top of the Gaia OS. The application framework [69, 131] is a component architecture based on the Model-View-Controller [88] user interface paradigm. The framework allows the partitioning of applications over groups of devices that can coordinate where the input is taken from, how the state of the application is presented, and where the application processing takes place.

The application framework uses four main components: the model, presentation, controller, and coordinator. The model implements the application logic. It also provides methods for storing and synchronising the application state, as well as providing an interface for accessing the functionality of the application. The presentation component provides a representation of the application state, for instance, by use of a graphical or audio representation, or a variation in the temperature or lighting in the environment. In any case the representation provided by the presentation component is any external effect to the physical environment that can be perceived by users. The controller alters the state of the application through mapping sensor inputs onto the model's interface. Examples of controllers may be a keyboard or a mouse but may also include changes in context. The coordinator manages the above three components. The model, presentation, and controller provide the main building blocks of application while the coordinator provides the application's meta-level functionality.

How these components are arranged to form an application is achieved independently of any particular active space through the use of generic application descriptions. The application framework uses two types of description files: application generic descriptions (AGD), and application customized descriptions (ACD). The AGD provides a generic template for an application that is independent of any active space. It describes the components required to compose the application including the name, type, and number of instances needed. The ACD is a customised description of the AGD that fits the resources of a particular active space. The ACD is defined using a high level scripting language called LuaOrb [31]. LuaOrb is based on the interpreted language called Lua [74]. The ACD is generated by the active space if one is not already available from the user. Both the ACD and AGD are stored in the

user's context file system and also provide the means of storing session information for the user.

When a user enters an active space the presence service detects the user. The presence service publishes an event to notify other components of the user's presence. The space repository receives the event and retrieves the user's profile and stores it locally. The context file system at the same stage locates the user's data and makes it accessible to the components of the active space. Other user devices can also be registered at this time. Any sessions activated by the user, or by the context requirements of the applications are then mapped to the active space. It is then possible for users to manipulate the inputs and outputs of applications to suit their needs.

The application framework provides Gaia with a flexible approach to structuring applications. By separating the inputs, outputs, and application logic of the application it allows Gaia to dynamically build applications to suit the available resources of an active space. It also serves as a means of adapting user sessions to any active space that a user may enter. The scripting mechanism used in the AGD and ACD also aid the rapid development of applications and help the incremental growth of the active space. However, there is an assumption that the components required to make the application will be available in all the active spaces. While this may the case for a large majority of components it may not be for all. In these cases the applications are unable to restart if the key components are not available. This subsequently causes user sessions to be inaccessible.

2.3.7.4 Summary

The Gaia project [134] has developed an infrastructure that supports the creation of user-centric, resource-aware, context-sensitive mobile applications. Mobile users are able to seamlessly interact and configure their physical and virtual environments in any of the active spaces they enter. Sessions allow data and applications to move transparently with users from one active space to another. On a user entering an active space the user's sessions are automatically mapped to the resources in the space. This ensures that data and applications are always available to users.

2.3.8 Portolano - One.World

The aim of the One.World project [60, 61, 62] is to develop applications that can continually adapt to a changing computing environment while still supporting the activities of users as they move through the physical world or switch devices. To achieve this the One.World project has focus on developing a system with services that ease the development of applications in managing the constant changes in the environment. In taking this approach One.World seeks to expose applications to change so they may implement their strategies to handling the transition.

2.3.8.1 The One.World Approach

Grimm et al. [60, 61, 62] identified that a key challenge in building pervasive computing applications is the highly dynamic nature of the environments in which they operate. They observed that for applications to function properly within such environments they must be able to continuously adapt to the movement of users and the changes in the environment. To accomplish this they defined three requirements that should be satisfied. In the first requirement, they advocate that as users move through the physical environment the execution context of their applications changes all the time. To avoid users having to manually configure applications they believe it is necessary to expose applications to contextual change to allow them implement their own strategies for managing the transitions. The second requirement defines the necessity to allow the dynamic composition of components to the point where they can just plug together without the intersession of users. Grimm et al. believe that this is necessary to avoid the impracticability of asking users to manually perform the composition. Lastly, to support collaboration of users it is necessary to make easy to access saved information and to be able to share information between components.

2.3.8.2 The One.World Architecture

The One.World architecture is shown in figure 2.8. Grimm et al. uses the four foundation services in the One.World architecture to directly address the requirements of change, dynamic composition, and the pervasive sharing of data. The system services, which are based on the

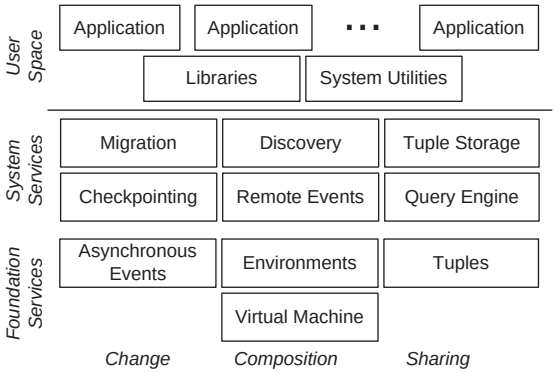


Figure 2.8: One.World architecture.

foundation services, provide additional services. These act as the main building blocks for applications in the One.World architecture. Grimm et al. organise the One.World architecture using a user/kernel split with the libraries, system utilities, and applications running in user space while the foundations services and system services are in the kernel.

The virtual machine, which in this case is a Java virtual machine, is used to provide a common execution platform to ensure that, at the lowest level, applications and devices are composable. Tuples are used to define a common data model for sharing information among applications. Tuples in the One.World architecture are records with named fields and are self-describing in that applications can dynamically determine the fields and their types. In using a common data model Grimm et al. not only eases the sharing of information but also encourages interoperability between arbitrary components. All communication between components in the One.World architecture, whether local or remote, is achieved through the use of asynchronous events. The exchange of events is achieved through the import or export of event handlers by components. The approach exposes applications to changes in the environment allowing them to adapt their behavior accordingly. The mechanism also helps to maintain loose coupling between applications and to increase system reliability by providing a system capable of handling component failure in a robust manner.

Environments are the main mechanism used to structure and compose applications within the One.World architecture. They act as containers that serve to store tuples, application

components, and other environments. The environments form a hierarchy with a single root on each device. Applications are required to have at least one environment to store application state though may contain several nested environments. All outer environments in the One.World architecture have total control of the environments they contain and are able to interpose on their interactions with the kernel and the outside world. The mechanism allows applications to be dynamically composed through the nesting of one environment within another. It also makes it possible to adapt and change the behavior of applications to suit different situations. The functionality of applications can also be extended in an incremental fashion using this mechanism. For example, the provision of additional security mechanisms, or a replication service.

System services build upon the foundation services to provide additional functionality for meeting specific application needs. For instance, the query engine allows applications to search tuples using filters. The structured I/O service lets applications access stored tuples in environments. The operations - put, read, query, listen, and delete - used in the structured I/O service are atomic and can also be grouped into transactions to provide one atomic unit. To use structure I/O applications bind to the environment's tuple storage and then perform operations on the bound resource. If the tuple storage is remote it instead binds to the network endpoint for the environment's tuple storage. In either case the bindings are controlled using leases [59]. The approach limits the time applications have access to environment's tuple storage while also allowing the One.World system to detect failure and to recover from it. Together, the query engine and the structured I/O service to simplify the access to data for applications.

Asynchronous events form the basis of communication in the One.World architecture. The distribution of these events across the network is done using the remote event passing service. To use the service components need to export event handlers under symbolic descriptors to the system. Interested parties, using the discovery service, can then locate and disseminate events to these components. The discovery service provides One.World applications with a number of options. For instance, it is possible to determine the binding time. This allows applications to determine when discovery is performed; early binding allows applications to first find

the resource and bind to it before sending a number of events, while late binding combines the discovery and event routing in one operation. The second option allows applications to disseminate events to a single matching resource, or multicast the event to all matching resources. In designing these facilities the goal of the One.World architecture has been to provide a flexible approach that could support a rich set of communication patterns. However, while this is achieved, there are a number of concerns with how the system scales. Currently, the discovery service is centralised, in that, one device is elected by the One.World devices to act as the discovery service for the network segment. There does not appear to be any bounds limiting the size of the pervasive computing environment that it serves and it potentially, as [62] indicates, will not be able to service requests in an acceptable time.

The checkpointing service is used to capture the execution state of an environment tree. It is stored as a tuple where it can be retrieved at a later stage by the One.World architecture and restored so that it can then resume normal execution. The migration service gives One.World the ability to move or copy an environment from one device to another. Together, the checkpointing and migration services help to provide application persistency and a method for recovering from major failures such as a device's batteries running out. The migration service also allows components to migrate with users as they move through the physical environment.

However, while the data-centric abstractions provided by the One.World architecture help construct pervasive computing environments there appears to be no high-level abstractions for sensors or actuators that can be used by applications to tailor their behavior and integrate into the physical environment. This has the affect of hindering the physical integration of applications and devices into the environment. It should also be noted that the One.World architecture does not explicitly provide security mechanisms for controlling access to environment's tuple storage or determining the propagation of the events.

2.3.8.3 Building Applications with One.World

Developing pervasive computing applications with One.World is centered on around the programmatic structure of the environment. Applications in One.World are required to have at

least one environment to store application state but may span several nested environments if required. The environment are used to store tuples, other application components, and to receive and send events. Access to threads, network sockets, and files are restricted and applications are required to only use the abstractions provide by the One.World architecture. The execution of applications is event-driven. Graphical user interfaces can be developed with Java's Swing toolkit which are able to run on top of One.World. However, as Grimm et al. note [61, 62] there are some limitations to developing pervasive computing applications with One.World. For instance, an application requires a class to access a tuple. This is a problem when environments are migrated to devices where the class cannot be retrieved resulting in the tuple being inaccessible to the application. Also, the use of Java's Swing toolkit for graphical user interface posed problems for Grimm et al. when migrating applications to devices with different screen characteristics.

2.3.8.4 Summary

One.World have developed an approach that focues on adapting applications to meeting the demands of users in highly dynamic environments. To manage and support applications Grimm et al. has developed the One.World architecture. In it environments are used as the main mechanism for structuring and composing applications while tuples form the basis of communication and data storage between the environments.

2.3.9 Other Pervasive Computing Projects

The projects and initiatives detailed in the previous sections show the diversity of approach used in tackling the challenges posed in developing pervasive computing environments. In providing a full picture of the current state of the art there are a number of other projects worthy of mention. The following paragraphs outline their contributions.

The EasyLiving [146] project, at Microsoft, has been researching a number of technologies for pervasive computing. Their basic infrastructure uses an agent-based system called InConcert, and a spatial model that defines geometric relationships between entities [21, 22]. The EasyLiving project uses the relationships between entities as the primary means of un-

derstanding the behavior of users in the space. By tracking certain objects and users in the space the EasyLiving project looks to adapt the behavior of applications to meet the expectations of users. The project relies heavy on a number of vision systems [89, 26] to track and recognise users and objects in the environment. They also use another system called RADAR [7] to track users. RADAR uses the strength of IEEE802.11 signals to determine the location of users.

MUSE [30] is a middleware architecture for pervasive computing environments being developed at the UCLA Multimedia Systems Laboratory. The MUSE infrastructure uses Jini [104] based services in combination with Bayesian networks to fuse raw sensory information into context information. Two of the primary services within the MUSE infrastructure are the sensor service and the sensing service. The sensor service represents the different sensors in the environment and acts as the proxy between the physical sensors and the other services and applications. The sensing service's main role is to provide other services and applications with context information. Using the Jini Lookup Service pervasive computing applications can find the appropriate sensing service and obtain the context information that is required.

eClass [1, 16], also know as Classroom 2000, investigates the impact of pervasive computing on education. The members of the Georgia Institute of Technology's Future Computing Environments Group (FCE) have built a prototype classroom environment. It captures the typical interactions that you might expect within a university lecture. The different streams of information are then integrated into a web interface which provides a summary of what happened in the lecture. The Zen* system provides the underlying infrastructure for eClass. It provides a number of tools for supporting lecturers in the preparation of their presentation, the capture of the presentation, and the integration of all captured materials into a form that then can be accessed at a later stage by students.

The Aware Home [85] research initiative is led by the members of Georgia Institute of Technology's Future Computing Environments Group (FCE). The have built a house, called the Aware Home, which is used as a test bed for developing technologies and techniques for enabling the creation of pervasive computing environments. Their research ranges from the fundamental sensor technologies, to middleware for building context-aware applications, to a

varied array of pervasive computing applications aimed towards helping the elderly. Dey's Context Toolkit [44, 43] provides the underlying infrastructure for many of the applications used in the Aware Home. The Aware Home initiative developed a number of pervasive computing applications. Some, as mentioned in [111], support the elderly in the home while others are based on more traditional application scenarios. For instance, the Digital Family Portrait [112] reconnects family members by providing a qualitative sense of a distant relative's well-being while striking a reasonable balance between privacy and the need for information.

ParcTab [164] was one of the first pervasive computing systems to be built. It was developed at Xerox PARC to explore the capabilities and impact of pervasive computing in an office setting. The system consists of various mobile and stationary devices such as the ParcTab, ParcPad, and the LiveBoard. The ParcTab is a small handheld device. The ParcPad is a tablet size unit and the LiveBoard is a large electronic display. Devices were able to communicate wirelessly through an infra-red network and could also be located to a particular room. The project paid particular attention to the networks, device technologies, and human computing interaction used in developing these types of environments. The work done at Xerox PARC has provided the inspiration for most of the projects and initiative mentioned in previous sections.

2.4 Summary and Conclusions

Weiser's vision of pervasive computing ignited a field of research that is now only starting to get to grips with the concepts outlined in his seminal paper [167]. The beginning of this chapter introduced his influential vision, a vision that realised the world of computing needed to move away from the conventional concept of the personal computer to an era where physical spaces would be composed of hundreds of computational devices that would be so interwoven into the fabric of everyday life that they become indistinguishable from it. In general, the challenges facing pervasive computing are interrelated with Wieser's central theme of making computers disappear from our awareness, ensuring that they can remain in the background of our society without requiring our conscious intervention in maintaining their presence.

To gain a greater understanding of the challenges facing pervasive computing the chapter

presented a number of research projects and reviewed their efforts in overcoming the hurdles posed by pervasive computing. The projects surveyed reflected current trends in pervasive computing research while also showing the diversity of approaches used in overcoming these challenges. The research surveyed lead to a number of observations that have formed the basis of this thesis. The observations identified were as follows:

- The literature suggests that the majority of pervasive computing systems can be characterised as conceptually centralised infrastructures that coordinate the resources of a specific geographical location. In some cases users are able to relocate their work from one environment to another. In contrast, chapter 3 presents a highly decentralized method for organising the components of a pervasive computing environment that supports spontaneous interaction of entities and the ad-hoc assembly of environments.

- The current state of the art would indicate that particular focus has been paid on understanding the challenges of developing pervasive computing environments in a number of well defined cultural boundaries. For example, meeting rooms, or offices are some prime examples. The systems supporting these environments are typically tailored to suit the devices - large mount displays, laptops, PDAs - and to meet the requirements of developing applications in these types environments. In chapter 5 the thesis looks to define a framework that supports the ad-hoc composition of everyday objects not normally considered in the literature. Providing a framework that allows pervasive computing environments to emerge from physical spaces as pervasive computing objects accumulate in the environment.

- Research has shown that context information is important in adapting the behavior of pervasive computing systems. This fact is used in chapter 3 to create a model where the context information of a local environment is used to manage the behavior of entities.

- Related research suggests that appropriate software architectures ease the development and deployment of pervasive computing applications. In chapter 4 and 5 the thesis looks to advance state of the art by presenting an alternative framework that supports the self-organisation of pervasive computing environments and promotes the autonomy of

entities. A high-level scripting language that simplifies the implementation and deployment of applications by allowing for the incremental construction and improvement of solutions.

- Research indicates that decoupled communication models have fewer dependences between components allowing the overall system to be less fragile and more stable to disturbances in the environment. This type of communication model is used in chapter 5 to ensure that robust system behavior can be achieved.

Chapter 3

A Stigmergic Model for Pervasive Computing

The chapter motivates the design of a framework intended to provide support for developing pervasive computing environments. The framework supports a highly-decentralised method of organising the components of a pervasive computing environment that allows spontaneous interaction between components and provides robust system-wide behavior. More precisely, the framework has been designed to provide developers with a convenient method of developing pervasive computing applications that eases deployment and supports the incremental growth of the environment. The inspiration for the approach stems from nature and the observations made by Grassé on how social insects coordinate their actions using indirect communication via the environment, a phenomenon known as stigmergy [58]. The design of the framework exploits these observations to allow the self-coordination of pervasive computing environments thereby promoting the autonomy of entities. To develop an understanding on how such an approach could work, this chapter describes a model based on the concept of stigmergy that underpins this framework for pervasive computing.

The chapter begins by presenting a set of requirements used to influence the design and implementation of the framework. The requirements are drawn from an analysis of the related work presented in chapter 2. This analysis highlights a number of limitations in current research when considering the ad-hoc composition of pervasive computing environments. Sec-

tion 3.2 introduces the natural phenomenon of stigmergy. It describes how social insects are able to use the environment as an indirect mechanism for coordinating their activities. The mechanism has been shown to be a highly-decentralised method of coordinating collections of interacting entities. Section 3.3 investigates how stigmergy can be used to successfully address the requirements. Section 3.4 describes a model based on stigmergy that the framework supports. The final section provides a summary of the chapter.

3.1 Analysis and Requirements

This section draws on the observations made in chapter 2 to establish a set of requirements that can be used to create a framework for supporting the ad-hoc composition of pervasive computing environments. The primary objective of this analysis is to identify the core architectural features required of a framework to support the next generation of pervasive computing environments.

3.1.1 The Next Step for Pervasive Computing

Technology for pervasive computing is reaching a point where it is becoming possible to convert many everyday environments into interactive spaces. As can be seen from chapter 2 particular focus has been placed on environments such as office spaces, lecture theatres, and research laboratories. Typically, these interactive spaces have been designed from the ground up to support the anticipated needs of their users and to evaluate the technology deployed in the space. The environments are usually pre-installed and maintained over the period in which they are in use. However, as Edwards et al. [50] point out, it is unrealistic to expect all pervasive computing environments to be constructed in this manner. They believe that physical spaces are more likely to evolve accidentally into pervasive computing environments as technology is incorporated into the space. Kindberg et al. express a similar view in [86] where they argue that pervasive computing systems will tend to form accidentally over the medium-to-long term.

This would suggest that pervasive computing environments need to be assembled in a

more ad-hoc fashion than has previously been the case. The approaches highlighted in the previous chapter do not readily apply themselves to this form of development. They appear to be more conceptually centralised approaches that focus their efforts around coordinating the resources of specific geographical locations. For instance, in the Stanford Interactive Workspaces project [78] mediates all interactions for the iRoom environment are mediated through the iROS system. It is responsible for managing all the resources at that particular location. While users are able to bring their laptops and PDAs into the space and have them seamlessly and transparently integrated into the room they cannot operate outside of this environment unless the iROS system has been installed at that location. It does not allow the ad-hoc interaction and coordination of components at locations other than those that have been predetermined. The system is conceptually a centralised infrastructure that restricts components forming pervasive computing environments to locations where the system has already been installed. While this method may work well for developing pervasive computing environments from the ground up it would be less appropriate to composing environments in the more ad-hoc manner anticipated by Edwards et al. [50].

In this form of development pervasive computing environments emerge from spaces through the migration and accumulation of technology at a particular location. There is no master plan that guides the development or any expert overlooking the construction of the environment. It evolves through ordinary people moving and integrating new technology into a space. Where an environment emerges depends totally on how the occupants arrange the technology. Unlike Aura [56], Gaia [134], or some of the other projects mentioned in chapter 2 the installation of a pervasive computing system into a physical location is not a prerequisite for the environment to form or to operate at those locations. For these types of environments devices and applications need to be able to spontaneously interact at any time and at any place without having to mediate their behavior through a central authority at each specific location.

In allowing environments evolve in an ad-hoc fashion it makes it possible for them to emerge at hotspots of activity where users require and want to use them and so give an illusion that they are always available. Kindberg et al. [86] also observe that through the

inclusion of new technology or the rearrangement of what is already there that new usage models can be adopted by the occupants. The environment continuously changes and adapts to how those in the space use it and rearrange the technology. It is not bounded by the same constraints normally imposed on a system specifically designed to operate at a particular location; it changes as users move technology into or out off the space. Changes may occur slowly over a period of time or at a much faster rate depending on how the space is being used.

The system software for managing such an environment needs to be different to that used in AT&T's Sentient Computing Project [2, 65], the Adaptive House [108], Aura [56], or the other projects previously described in chapter 2. In these types of environments the components - devices, physical artifacts, software components, and services - that comprise the system must be organised in a highly decentralised manner. Unlike many of the systems presented in chapter 2 there is no central component in the environment to manage access to resources or coordinate how different components in the environment interact with each other or with those using the environment. The components are in fact the system and as such have to be able to spontaneously interact with each other to coordinate their behavior in a distributed manner. The environment can be though of as a collection of interacting components that through their ad-hoc interactions can form a pervasive computing environment capable of providing services for those occupying the environment.

3.1.2 Requirements for a Pervasive Computing System

In this section we identify the requirements that need to be satisfied in order to support the type of pervasive computing environments discussed in the previous section. The requirements are drawn both from this vision and from the observations made in chapter 2 on the challenges posed to pervasive computing and on the projects reviewed. The emphasis is on enabling infrastructure-free systems that provide a highly-decentralised method of coordinating collections of interacting entities, more over, the focus is on the middleware aspects of building such systems rather than other concerns of pervasive computing. In meeting this aim the following section outlines the core set of architectural features required of a framework

that is capable of supporting these types of pervasive computing environments.

R1: Support the physical integration of components into the environment. For the type of environments envisioned in the previous section as well as for many other pervasive computing environments it is important to abstract the complexities of dealing with the real world to ensure that components can easily be integrated into the physical environment. To achieve this goal it is necessary to provide high-level abstractions that allow components to sense and interact with the physical environment without the difficulties of dealing with low-level devices such as sensors or actuators. The framework needs to provide mechanisms for retrieving data from sensors and fusing it together to produce a reliable view of the environment. In so doing, it must alleviate the uncertainty typically associated with dealing with unreliable sensors. It must also cope with the highly dynamic nature of pervasive computing environments where sensors and actuators can be inserted into, or removed from the system at any time. These concerns need to be addressed if the framework is to support the physical integration of components into the environment. Overall, the challenge is to deliver an appropriate level of abstraction that allows components to gain enough understanding of the real world while ensuring there is sufficient low-level support for observing and manipulating the environment.

R2: Support the autonomy of components. A key characteristic of these types of environments is the autonomous nature of the components of which they will be comprised. Each component is an independent entity with the ability to move through the environment unrestricted. It does not depend on other components to operate and is responsible for managing and controlling its own behavior within the environment. While each component is self-regulating its behavior can be influenced by those occupying the space and the subsequent behavior of other components. The autonomous nature of components ensures that the environments envisaged in section 3.1.1 can emerge at any location as their movement is not restricted to a particular location or their behavior curtailed by other components.

R3: Support spontaneous interoperability between arbitrary components. From the discussion presented in section 3.1.1 it is clear that the environments envisioned are developed in an accidental manner over a period of time and not in any structured manner. These type of systems evolve as those using the space introduce or remove components from the environment. Over the life time of the environment it can be expected that a wide range of heterogeneous components will be integrated into the environment or will move through it on a regular basis. It can therefore be assumed the components comprising the system will have little prior knowledge of the other parts of the environment with which they will interact. Consequently, it is important that components be able to spontaneously interact with each other with little or no prior knowledge. To achieve this the framework needs to provide mechanisms that allow components to discover other parts of the environment or to develop common interoperation models to which components can comform and which assist them in interacting with one another in a spontaneous manner. Overall, the goal is to ensure all components can be seamlessly integrated into the environment in a manner that allows the system to grow organically and decay gracefully as new components are added and old ones upgraded or removed.

R4: Support the decentralised coordination of component behavior. Contained in these pervasive computing environments are large collections of autonomous components capable of spontaneously interacting with each other. To ensure a coherent environment can be formed it is necessary for these components to be able to coordinate their behavior in a way that provides a meaningful experience to users. A centralised approach is not a feasible approach to coordinating these types of environments as they are not structured around any particular focal point, whether that be a physical location, person, or single component. It is envisioned that these environments form from the coordination of an arbitrary collection of components and it is therefore necessary for the framework to provide a decentralised mechanism for coordinating the behavior of components within the environment.

R5: Provide a scalable solution. One of the features of pervasive computing and of the environments presented in section 3.1.1 is the large number of components expected to be

interwoven into the environment. It can be assumed that, as these environments grow, the intensity of interactions will increase with the number of components and users inhabiting the space. Thus, the scalability of the system is of particular concern especially when considering the limited resources - bandwidth, energy, and computational power - found in these systems. A number of projects [56, 34, 78, 134] have solved the problem by bounding the environment to a well-defined physical space, such as an office or room, to limit their size. However, it is not possible to use this same approach to develop the type of environments presented in section 3.1.1 as they are not limited to or bounded by any particular room or physical space. Components are free to roam across the entire environment and to interact with any other part of the environment they find of interest. It is therefore conceivable that a pervasive computing environment could span a series of cultural boundaries that would have typically been used by other projects [56, 34, 78, 134] to scope the intensity of interactions. Hence, it is necessary for the framework to develop alternative mechanisms that can scope the interactions between components in a way which allows the system to scale but does not affect the overall experience of users.

R6: Ensure the robust behavior of the system. In pervasive computing it has to be recognised that failure of components occurs on a regular basis. The causes may be due to the fragile nature of some of the hardware used, to the limited life span of batteries, or the unreliability of some wireless networks. Whatever the reason it is necessary for the system to be able to absorb the underlying changes to ensure the environment can behave in a robust fashion. The framework must ensure that the failure of one component does not lead to a knock on effect of other components failing within the environment. There is a need to keep failure of components localised to the point where it has occurred and not allow it to spread. The overall goal is to ensure the pervasive computing system can handle disturbances in the environment without adversely effecting the operation of all the components in the space.

R7: Support incremental construction and improvement of solutions. In developing the environments described above it has to be expected that they will evolve incrementally over a period of time. It cannot be assumed that an environment of this nature can be de-

veloped or installed in one go. The framework used to support such an environment has to be designed in such a way that it will allow the incremental construction and improvement of solutions without adversely effecting the rest of the environment. To aid development and the deployment of components it is also necessary to separate the complexities of the underlying system with the compositional side of defining component behavior. This should also assist the rapid development and evolution of the environment.

R8: Mobility. With the expected migration of technology and the anticipated movement of users within the environments described in section 3.1.1, it must be assumed that there will be a high degree of mobility among pervasive computing environments of this nature. In these situations the components that form the environment and those that occupy it are continously moving through the physical environment reacting to and coordinating their actions with other parts of the environment. While the components are autonomous in nature and capable of spontaneously interacting with the other parts of the environment it is necessary to ensure mechanisms are in place to make the movement of users and components largely transparent to developers and that the spontaneous interoperability between arbitrary components is not effected by their mobility.

R9: Adaptability. There is a high probability that the mobility of users and of the components that comprise a pervasive computing environment will lead to a situation where the environment is continuously changing. This is particularly the case for the type of environments envisioned in section 3.1.1 where autonomous components roam across the environment. To overcome this situation it is necessary for components to be able to adapt their behavior to use whatever is available in the immediate environment. The framework needs to provide a means of allowing this adaption to take place by providing components with a view of the environment they can use to adjust their behavior. The challenge is to achieve this with little or no outside intervention.

R10: Security and Privacy. Privacy is of great concern to users and is a particularly complicated issue for pervasive computing and one that causes a great number of concerns

[40, 135, 86]. To allow devices and applications to move into the background of society they must be able to obtain information about users to anticipate what is required from them. This knowledge is extremely sensitive information that can be easily exploited against a user but is critical for the successful operation of a pervasive computing environment. There is a serious potential for the loss of privacy which may lead to users being dissatisfied and unwilling to participate in the environment. In such cases a user must be able to trust that the information will not be misused by the environment or ensure the mechanisms used to obtain the information maintains the anonymity of the user. The issue of privacy is also very much interrelated with that of security. Pervasive computing systems need to prevent the unauthorised use of devices to prevent their misuse and access to private data. For people the decision to grant access is made intuitively through indept understanding of the trade-offs of acceding to a request. Traditional methods of securing computing systems are not easily applied to decentralised systems such as those envisaged in section 3.1.1 or for pervasive computing in general. It is therefore necessary for the framework to develop methods that can determine the trustworthiness of users in these situations while still maintaining a comfortable balance between granting access and preserving the privacy of users.

3.1.3 Existing Support of Requirements

The review presented in chapter 2 of pervasive computing revealed a wide range of approaches used to support the development of pervasive computing environments. Based on the set of requirements identified in section 3.1.2 we investigate the extent to which state-of-the-art environments support these requirements.

While each of the projects supports a number of the requirements critical for developing the type of pervasive computing environments presented in section 3.1.1 there is currently no one system that provides support for all requirements. Table 3.1 illustrates the set of requirements for which support is required and evaluates whether the projects presented in chapter 2 are capable of satisfying them. In the table, a bullet (•) indicates support for the requirement, while a small circle (◦) shows limited support, and no circle or bullet dictates there is no support for the requirement.

Requirement		Aura	Ambient's Cooperative Project	The Adaptive House	Project Oxygen	AT&T Laboratories - Sentient Computing	Stanford Interactive Workspace Project	Gaia	Portolano - One.World	Easyliving	MUSE	eClass	The Aware Home	ParcTab
R1	Support the physical integration of components into the environment.	●	●	○	●	●		●		●	●		●	○
R2	Support the autonomy of components.						○		○					
R3	Support the spontaneous interoperability between arbitrary components.	●	●		●		●	●	●	●	●			
R4	Support the decentralised coordination of component behavior.													
R5	Provide a scalable solution.	●			●			●						
R6	Ensure the robust behavior of the system.				●	○	●		●					
R7	Support incremental construction and improvement of solutions.		○		○		●	●	●		●			
R8	Mobility.	●						●	●					●
R9	Adaptability.	●	○	●	●	○	●	●	●		○			○
R10	Security and privacy.	○					○							

=no support ○=limited support ●=supported

Table 3.1: Features provided by the state-of-the-art approaches to develop pervasive computing environments.

Chapter 2 has identified a number of projects capable of providing adequate high-level abstractions for sensing and interacting with the physical environment (e.g., Aura, Gaia, Easyliving). Other approaches provide mechanisms to support the spontaneous interoperability between arbitrary components (e.g., Stanford Interactive Workspace Project, Project Oxygen). Projects such as Aura and Gaia provide scalable solutions though not in a form which can be usefully applied to fully fulfilling requirement R5. A number of the other projects have developed methods that allow the environment to behave robustly in the face of disturbances (e.g., One.World, Standford Interactive Workspace Project). It can also be seen from table 3.1 that the majority of projects are capable of supporting the incremental construction and improvement of solutions to some degree, though it must be pointed out that this has not been achieved for the environments presented in section 3.1.1. The ability to adapt to a changing environment is supported by a number of the projects most notably by Aura and Gaia. The approach used by these projects also allows support for the mobility of users from one environment to another.

While some of the projects - Gaia, Aura, Project Oxygen - have managed to successfully fulfil the majority of the requirements they fail to implement two of the key requirements for developing the type of environments discussed in section 3.1.1, namely support for the autonomy of components and the decentralised coordination of component behavior. These are central to the development of the type of environments presented in section 3.1.1 and consequently have a knock-on effect to how the other requirements are implemented. It is also notable that the security and privacy of users is not addressed fully by any of the projects investigated.

3.1.4 Summary

From the observations made at the end of chapter 2 we identified what we consider to be the next step for pervasive computing. Section 3.1.1 outlined this proposal and described the type of pervasive computing environments we envision while section 3.1.2 provided the requirements necessary for building these types of environments. The last section investigated the extent to which the projects presented in chapter 2 have met the stated requirements. It was found

that while a number of the projects have addressed some of the main requirements they fail to implement the key requirements for developing the type of environments envisioned. With this mind, the next section investigates swarm intelligence techniques, in particular, the natural phenomenon of stigmergy with the aim of applying these techniques to developing a framework capable of satisfying the requirements stated in section 3.1.2.

3.2 Swarm Intelligence - Stigmergy

Natural systems such as flocking birds and colonies of social insects have inspired a number of research efforts in robotics [71], in pattern detection and classification [20], and communication networks [83]. The potential of these collections of simply-behaving entities was first observed by biologists when studying colonies of social insects. They noticed that through local interactions of individuals a colony of insects could produce complex collective behaviors at the colony level. It has also been discovered that the coordination of individuals' activities does not require any supervision and produces a system that is a highly decentralised and is largely self-organising. The mechanisms used to organise these types of systems and the collective behavior that emerges from them has become known as swarm intelligence. Bonabeau et al. [14] define swarm intelligence as the collective behavior that emerges from a group of social insects. In [12], Bonabeau et al. describe it as the attempt to design algorithms or distributed problem-solving devices inspired by the collective behavior of social insect colonies and other animal societies. Kennedy et al. [84] goes further by describing swarm intelligence as any loosely structured collection of interacting entities.

Biologists have observed that interactions between entities in these types of natural systems can occur either directly or indirectly. Flocking birds or shoals of fish are a prime example of natural systems using direct interaction to coordinate their activities. Birds use visual contact with their neighbours to stay together, to coordinate turns, and to avoid obstacles. Shoals of fish use a similar approach to swim in tight formation. Indirect interactions are more subtle. Entities use the environment to mediate their communication. Indirect interaction occurs when an entity changes the environment and other entities respond to the modified environment in some way. This type of interaction is more commonly know as stigmergy.

3.2.1 Stigmergy

Stigmergy was first introduced by a French biologist called Grassé. He observed that social insects could coordinate their actions through the environment without having to directly communicate with each other, a phenomenon that he called stigmergy [58]. He also observed that through the local interactions of insects a colony-wide behavior could emerge. Beckers et al. [11] suggests that the origin of the word stigmergy is derived from the roots stigma 'goad' and ergon 'work', thus implying a sense of incitement to work by the products of work. However, Parunak [116] provides an alternative derivation and suggests that the term is formed from the Greek words stigma 'sign' and ergon 'action' and so therefore captures the notion of an entity's actions leaving signs in the environment that influence the subsequent actions of other entities. In any case the concept of stigmergy has been applied to a series of different fields since it was first discovered by Grassé in 1959. The literature would indicate that Grassé's definition of stigmergy has been interpreted and applied in a number of different ways as discussed in the following section.

Research would generally show that stigmergy is a form of communication used by entities to coordinate their activities. Mason [96] describes stigmergy as communication that is mediated through changes in the environment. Caro et al. [27] describe it as the indirect communication that takes place among individuals through modifications induced in the environment. Valckenaerks et al. [161] argue that it is a form of asynchronous interaction and information exchange between agents mediated by an active environment. D'Angelo et al. [38] define stigmergy as cooperation without communication. Beckers et al. [11] describe it as the indirect communication between agents through the sensing and modification of the local environment. Thus, the literature clearly indicates that stigmergy is an indirect communication mechanism that is mediated through modifications of the environment. Communication occurs when one entity changes the environment and another entity senses the modification and reacts to it.

Various research efforts have derived different understandings of what constitutes indirect communication between entities. In the context of social insects [12] indirect communication manifests itself through the deposits of chemicals - pheromones - in the environment or the

physical transformation of the environment as seen in the nest building of wasps. In the work by Mataric on autonomous agents [97, 98] indirect communication is based on the observed behavior, not communication, of other agents and its effects on the environment. Mataric refers to this type of communication as the modification of the environment rather than direct message passing between agents. Mason [96] has a similar view in that communication takes place through the environment rather than direct signal transmissions. In a taxonomy on agent coordination and cooperation Parunak et al. [117] view the information flow in stigmergy as the non-message interactions between agents. In general, the majority of research [71, 83, 11] considers that indirect communication is mediated through some form of environment.

Different varieties of stigmergy have been distinguished. Theraulaz et al. [160] describe two forms, quantitative stigmergy and qualitative stigmergy. Bonabeau et al. [12] also characterise stigmergy in a similar fashion though they also use the term discrete stigmergy interchangeably with qualitative stigmergy. In quantitative stigmergy the stimulus varies in a quantitative manner. A prime example is the construction of termite mounds where workers add pheromones to soil pellets they place on a growing mound. The pheromone fields and gradients that subsequently appear influence where individuals place their pellets, which is normally at the highest concentration of the pheromone. By placing the pellet at an already high concentration of pheromone, termites create a positive feedback mechanism that stimulates the construction of the rest of the structure. The structure initially emerges from the amplification of random deviations of pellet placement and stabilises through the negative feedback of the decay of pheromone in the environment over time. Systems using quantitative stigmergy are generally considered to be self-organising as they typically subscribe to the four basic ingredients [12] that self-organisation relies on: positive feedback, negative feedback, amplification of fluctuations, and multiple interactions between individuals.

Qualitative stigmergy differs from quantitative stigmergy in that it is based on a discrete set of stimuli types. In this case insects change their behavior in response to qualitatively different stimuli. For example, an insect would respond to a type 1 stimulus with behavior A and a type 2 stimulus with behavior B. The nest building of social wasps [12, 160] is an example of qualitative stigmergy used in nature. Nests are built up from wood fibers and

plant hairs and cemented together with salivary secretions. This is then moulded by the wasp to form the different parts of the nest. The process starts with the construction of a pedicel, which attaches the nest to some physical structure. The wasps continue by building two cells on top of the pedicel and from there start to add cells to the outer circumference of the nest. After a row of cells has been completed the wasp starts on the next row. The decision to build a new cell at a location is influenced by the number of adjacent cell walls that have previously been constructed. A wasp receives qualitatively different stimuli in a location where there is one cell wall to a location where two or three cell walls are available. The probability of a wasp adding a new cell increases with the number of adjacent cell walls already in place. The process forms closely packed parallel rows of cells that have a symmetric radial or bilateral shape around the initial cells. Bonabeau et al. [12] argue that due to the stimuli being qualitatively different that it is not possible to amplify the stimulus through positive feedback and so self-organisation is difficult to achieve through this form of stigmergy. Camazine et al. [25] and Bonabeau et al. [12] suggest that this type of stigmergy is a form of self-assembly, in that, the building elements, mediated by the actions of individual entities, self-assemble to form the appropriate structure.

Other research efforts have classified stigmergy in a variety of different ways. Holland et al. [70, 71] define two forms of stigmergy: active and passive. Active stigmergy is when an entity's behavior is affected by a qualitative or quantitative effect in the environment. Passive stigmergy occurs when a previous action affects neither the choice nor the parameters of subsequent actions but does effect the final outcome. Holland et al. describes this type of stigmergy as being very close to purely physical situations where a constant force changes the environment in such a way as to change the force's future effect on the environment. An example of this type of stigmergy is the formation of sand dunes through air moving over their surface, and meandering rivers via the water which flows through them. Another example, used by Stone et al. [150], is if one agent turns off the main water value to a building, the effect of another agent turning on the kitchen tap is altered.

White et al. [171] and Brueckner [19] have an alternative classification of stigmergy based on Wilson's observations [173]. They define two types of stigmergy: sematectonic and sign-

based. Sematectonic stigmergy is where entities guide their work through the evidence of previous work accomplished. Sign-based stigmergy occurs when marker deposits are excreted into the environment to influence the subsequent behavior of entities. The classification is very similar to the Theraulaz et al. [160] definition of qualitative and quantitative stigmergy. To aid understanding of the different classifications of stigmergy table 3.2 provides a summary of the points made.

		Categories of Stigmergy.		
Classfications	Theraulaz et al. [160] & Bonabeau et al. [12]	quantitative	qualitative	
	White et al. [171] & Brueckner [19]	sign-based	sematectonic	
	Holland et al. [71]	active		passive
The type of stimlus involved.		Individuals modify their behavior to a stimulus that varies in a quantitative manner.	Individuals change their behavior to qualitatively different stimuli.	When a pervious action effects neither the choice nor the parameters of subsequent actions but does effect the final outcome.
Self-organisation involved.		yes		
Self-assembly involved.			yes	
Example.		The construction of termite mounds.	The nest building of social wasps.	The formation of sand dunes through the movement of air over the surface.
Type of interactions between individuals.		Indirect communication mediated through modifications of the environment.		

Table 3.2: A summary of the classifications used to categorise the different forms of stigmergy.

In developing these types of systems Holland et al. [71] suggest that there is a minimum set of requirements that entities and the environment need to process if they are to support

stigmergy. They are as follows:

- Entities should be able move through the environment.
- Entities need to be able to act on the environment.
- The environment must be able to be changed locally by entities.
- Changes in the environment need to persist long enough to effect the behavior of entities.

Stigmergy can be defined as an indirect communication mechanism that allows entities to structure their activities through the local environment. It has also been shown to be a primary ingredient in coordinating the complex behaviors seen in social insects. In the next section the thesis explores how stigmergy has been use in computer-related research.

3.2.2 Research Inspired by Stigmergy

The potential of social insects has not gone unnoticed. Several research initiatives have looked to harness the coordination mechanisms used by these types of natural systems to develop techniques and algorithms for solving a range of computer-related problems. The following section highlights a number of them.

One of the classical behaviors of some species of ants is the trail-laying and trail-following they use when foraging for food. Ants deposit a pheromone on their way back from a food source. Foraging ants follow such trails. The process has been experimentally shown to be self-organising [41] and to optimize on the shortest path from the nest to the food source [57]. The phenomena uses quantitative stigmergy to coordinate the behavior of the colony. The ant foraging behavior inspired a problem solving technique called ant colony optimization (ACO) [47, 45]. The ACO algorithm creates a colony of artificial ants that cooperate to find solutions to difficult discrete optimization problems. It has been applied to the travelling salesman problem [46], routing in communication networks [171, 27, 83], vehicle routing [23], and the quadratic assignment problem [95]. Both Dorigo et al. [47] and Bonabeau et al. [12] illustrate the use of the ACO algorithm in a number of other examples.

The concept of stigmergy has also had a significant influence on the area of behavior-based robotics. Beckers et al. believe that the "fit between stigmergy and behavior-based robotics is excellent" [11]. They argue that conventional robots were too slow to cope with the interactions of multiple robots and too expensive to build in large numbers with which to experiment. They suggest that by using behavior-based architectures, such as Brooks' subsumption architecture [15], in conjunction with stigmergy a more feasible solution can be obtained. To test their approach they built a number of small robots (21x17cm). The robots were equipped with two infra-red sensors, a pressure sensitive gripper for pushing objects around, and a very simple internal behavioral script that manages the behavior of the robots. The task they choose to use to investigate how such a system would behave was based on the corpse-gathering behavior of ants. The robots would push pucks (4cm in diameter, and 2.5cm in height), which were evenly distributed in the environment, into a single tight cluster. After carrying out a series of experiments varying the number of robots used and robot configuration Beckers et al. observed that over a period of time the robots where able to form a single cluster of pucks. Similar work was also produced by Holland et al. [71]. He investigated how simple robots could be used to sort different coloured frisbees into clusters. Werger also used the concept of behavior-based robotics with stigmergy to particularly good effect. He developed a team of robots to compete in RoboCup97 a robotic football competition held in Japan [170, 169]. His team "The Spirit of Boliva" was entered in the mid-size robot league, where no global data is available between robots and only local sensing is permitted. Even so the Spirit of Boliva team was able to display planned for, interactive team cooperation. In the end Werger's team was the only team to have defeated one of the champions and the only team not to concede any goals.

Stigmergy has also been used in agent-based systems to coordinate the actions of agents in solving particular problems. For example, Brueckner and Parunak [20] use the concept of stigmergy to find global patterns across spatially distributed real-time data sources. Their motivation was to develop a system for the early detection of large-scale bio-terrorist attacks on the civilian population. They deployed a large population of simple mobile agents in a distributed network of processing nodes. The agents would move across the nodes in search of

places with a strong spatial gradient in the data. Agents collectively coordinate their search through the deposits of pheromones at each node. The pheromones serve as a means of attracting other agents to potentially interesting patterns which in turn reinforces the presence of the pattern. Valckenaers el al. [161] have also used an agent-based system coordinated by stigmergy to control manufacturing processes. The software architecture uses a world model of the manufacturing process that maintains the pheromones in the environment and the location of agents in the system. System behavior - control of the manufacturing process - emerges from the activities of the agents reacting to pheromones in the environment. In [63], Gulyas proposes another system for coordinating the behavior of mobile agents using stigmergy. Again, agents can move from one node to another writing messages - cues - to a blackboard for other agents to act upon. In the same way that pheromones evaporate over time, the messages placed in the blackboard gradually disappear. Mamei and Zambonelli [94] have also relied on the concept of stigmergy to coordinate collections of interacting agents in an active environment. Their approach is centered around a tuple space mechanism which acts as the environment for the agents. The agents can observe the environment by reading the tuples and can modify it by inserting new tuples into the space.

3.3 Using Stigmergy in a Pervasive Computing Environment

The literature shows a number of biologically-inspired research efforts which have used stigmergy as a basis for tackling various computer related problems. The mechanisms used to organise these types of systems and the collective behavior that emerges from them is also an appealing construct for pervasive computing. In the following sections this thesis adopts the principles of stigmergy to design a system that can be used to provide a highly decentralised method of organising the components of a pervasive computing environment. The section first examines whether such an approach can produce a system capable of supporting the type of environments previously discussed in section 3.1.

3.3.1 Collections of Interacting Entities

The idea of simple insects, with little memory or ability to exhibit any real intelligence, maps well to pervasive computing where devices with limited resources are spread across the environment. The large number of devices expected to be deployed into our society matches the scale at which these colonies of social insects work. The constant interaction between components of a pervasive computing environment also ties in neatly with how social insects interact with each other. It would appear that a colony of social insects are in many ways very similar to a pervasive computing system where large collections of interacting entities roam across the environment. The question is whether the same coordination mechanisms used by social insects, especially stigmergy, can be harnessed in a similar way to manage the components of a pervasive computing system over the life time of the environment. If this is indeed possible it can be expected that the same highly decentralised mechanisms used to coordinate colonies of social insects can be applied to developing pervasive computing environments.

3.3.2 Autonomy and Decentralised Coordination

In applying the principles of stigmergy to pervasive computing it should be possible to harness the same mechanisms of coordinating large collections of interacting entities as social insects utilise, and in so doing, provide a predominately decentralised method of organising and controlling groups of components in a pervasive computing environment. This is achieved by components moving through the environment and using local interactions, mediated via the environment, to coordinate their actions with other parts of the system. As with the social insects presented in section 3.2 the components of a pervasive computing system modify their local environment to influence the subsequent behavior of other components, a natural phenomena which has already been shown to be highly decentralised and largely self-coordinating in colonies of social insects. It should also be stressed that individual entities need have no particular problem solving knowledge, and that coordinated behavior emerges due to the actions of the society. In a pervasive computing environment it can be expected that the coordinated actions of components would be capable of providing services for those occupying

the space.

In taking this approach it is possible to meet one of the main requirements highlighted in section 3.1.2 for developing the type of pervasive computing environments discussed in section 3.1.1; supporting the decentralised coordination of component behavior. An underlying feature associated with these types of decentralised systems is that the entities control and manage their own behavior within the environment and are as such autonomous. This can be seen in social insects such as a colony of ants where each individual manages its own behavior in relation to the rest of the colony. Such behavior would fulfill the requirement to support the autonomy of components within a pervasive computing environment as described by requirement R2 - autonomy of components - in section 3.1.2. It can also be argued that the inherent mobility of entities in a stigmergic system is sufficient to meet the main aspects of requirement R8 - mobility.

3.3.3 Spontaneous Interoperability

In a stigmergic system the environment acts as a shared medium through which entities communicate. Each entity manipulates the local environment in a way that is eventually recognisable to other entities in the surrounding area. For some social insects this takes the form of pheromone deposits, for others, like the wasps described in section 3.2.1, it is achieved through the construction of cell walls. In either case the alterations performed by the entities are universally understood by all entities involved making it possible for them to spontaneously interact with each with little or no prior knowledge of the other entities. Used in pervasive computing the environment should also provide a common interoperation model capable of allowing components to interact in a spontaneous manner. It provides a common interface that all components must use if they are to coordinate their actions. For components to use this interface they must to able to perform two basic operations. First, each component needs be able to sense their local environment to detect any stimulus that might effect their behavior. Secondly, components must be able to change the state of the local environment. These changes need to persist long enough in the environment to effect the subsequent behavior of other components, and typically, may consist of components displaying

information, or physically altering the environment in some way. The environment is acting as a common shared service to all components making it possible to seamlessly integrate any arbitrary component into the interactive environment with minimal human intervention. It allows for the impromptu interoperability that Edwards et al. [50] advocates is necessary for the successful operation of a pervasive computing environment. Using such an approach makes it possible to fulfill requirement R3 - spontaneous interoperability - for a pervasive computing environment.

3.3.4 Robust, Adaptable Environments

One of the advantages of using techniques based on stigmergy in pervasive computing is that it allows a system to harness the same robust, self-coordinating mechanisms observed in colonies of social insects. It provides a very flexible approach of adapting a system to a changing environment. These properties have been observed throughout the literature. For example, Theraulaz and Bonabeau noted the 'relative stability of coordinated algorithms with respect to perturbations' [159]. Also, in the work performed by Beckers et al. they observed the system was 'able to cope with occasional robot failure' [11].

The robust nature of these types of systems is an attractive property for pervasive computing where the system is generally considered to be highly dynamic and unpredictable. In particular, the indirect communication mechanisms used by social insects provide a means of decoupling components within the system. This is achieved as communication between social insects is mediated through the local environment and not directly between them. Applied to pervasive computing it leads to a situation where there are fewer dependences between components making the overall system less fragile and more stable to disturbances in the environment. It keeps the failure of components localised to the point where this has occurred and prevents it from spreading to other components in the environment. This is possible due the autonomous nature of components and the level of indirectness provided by stigmergic systems.

It must also be noted that the flexibility provided by stigmergic systems and ability for them to adapt to changing environments is one of the more interesting characteristics provided

by these type of systems for pervasive computing. There ability to adapt can be seen in a number of adaptive routing protocols used in communication networks [171, 27, 83] or particular stigmergic models [13, 12] based on task allocation or the division of labor observed in some species of social insects. These systems can be seen to adapt both at the level of the colony and that of each of the individuals in the colony. To achieve the equivalent functionality in pervasive computing each component needs to mimic the same type of behavior as those performed by social insects. In this case they have to be able to observe the modifications being made in their local environment and have the ability to act upon them. At the same time they must also be able to make changes to the local environment. By using such an approach it is possible to fulfill requirements R6 - robust behavior - and R9 - adaptability - specified in section 3.1.2.

3.3.5 Simple, Scalable Solutions

In observing a colony of social insects you can't help but notice the scale at which these organisms work. A swarm of raiding army ants (Eciton burchelli) may contain up to 200,000 workers raiding a dense phalanx 15m or more wide [12, 25]. What is also evident is each individual is simply constructed, in that, they have no particular problem solving knowledge and are simply following a basic set of instructions that allow them to react to the environment. Beckers et al. [11] commented on this when they performed a series of experiments with collections of robots they built to investigate the use of stigmergy. Holland et al. [71] also noted the simple set of instructions used to control the robots in their experiment on stigmergy. Parunak [116] also notes 'the logic for individual agents is much simpler than for an individually intelligent agent'.

All interactions in a stigmergic process are mediated through the local environment. By using this fact in a pervasive computing system it is possible to ensure that a scalable solution can be obtained. In these cases entities are only interested in observing the state of the environment local to them as it is only this part of the environment that influences their behavior. Applying the same process to pervasive computing would severely reduce interactions with distant locations, therefore, allowing the system to scale more gracefully. Both

Satyanarayanan et al. [135] and Kindberg et al. [86] have identified the usefulness of applying such an approach. Satyanarayanan et al. proposed the idea of localised scalability. They recognised that the density of interactions with any particular entity falls off as one moves away from it. They suggest that this fact can be used to help provide scalable solutions for pervasive computing. Kindberg et al. draw a similar conclusion with the boundary principle. They argue that system designers should divide the world into physical environments that demarcate their content.

It should also be noted that the simple nature of individual entities implies there is very little processing power required for them to operate. For pervasive computing this would indicate components are able to run on extremely small platforms such as Smart-Its [72] or allow embedding several components into more powerful devices without raising issues of scalability. Mason [96] also points out that stigmergic systems use minimal communication. Used in pervasive computing the communication required to operate a successful environment is significantly reduced, aiding the scalability of the environment and utilising the limited resources in a more efficient manner. It is clearly evident by applying the principle of stigmergy to the design of a pervasive computing framework that it should be possible to fulfill the requirement R5 - scalable solution - outlined in section 3.1.2.

3.3.6 Incremental Construction and Improvement of Solutions

It can be argued that the autonomous nature of individuals and the manner in which each individual is simply constructed allows such systems to be totally extensible, in that, new entities can always be added and updated when necessary. This is possible due to the loose coupling associated with entities of stigmergic systems and their ability to adapt to a changing environment. Applied to pervasive computing the autonomous nature of individual components and the loose coupling between them ensures a pervasive computing system can always grow and decay with the addition of new components and the upgrade or removal of old ones. It allows components to be developed separately and to be installed into the environment when they are ready. Over the life time of the environment new components can always be inserted into the system without causing difficulty to the rest of the environment. More

importantly, their ability to adapt will ensure new components can be incorporated into the operation and functionality of the existing environment. The simple nature of entities in stigmergic systems should also aid the rapid development and evolution of the environment. As noted in the previous section they are simply constructed, in that, they generally follow a basic set of instructions to determine how they react to the environment. This characteristic can be exploited in pervasive computing to provide a programming interface that can aid the rapid development of components. Harnessing these types of abilities makes it possible to construct a pervasive computing environment incrementally over a period of time. In so doing, it fulfills requirement R7 - incremental construction - specified in section 3.1.2.

3.3.7 Providing an Appropriate Level of Abstraction

In section 3.1.2, requirement R1 - physical integration - is concerned with abstracting the complexities of dealing with the real world to ensure that components can easily be integrated into the physical environment. The aim is to provide high-level abstractions that give components enough understanding of their environment while ensuring there is sufficient low-level support for observing and manipulating the environment. To address these concerns a pervasive computing system needs to provide a programming abstraction that allows components to sense and interact with the physical environment without the difficulties of dealing with low-level devices such as sensors and actuators.

In stigmergic systems large collections of entities move across the environment. Each of these entities would typically have the ability to sense the part of the environment that effects their behavior. They also have the ability to manipulate their environment in a manner that may influence the subsequent behavior of other entities. Applied to pervasive computing each component would use a number of sensors to detect the environment and have a series of actuators that would allow them to manipulate it. The use of stigmergy in pervasive computing imposes a certain structure on the system and each of the components in that system, however, this is not sufficient to provide the high-level abstractions sought by requirement R1 - physical integration. To address these concerns a pervasive computing system will have to provide additional support in the way of a programming abstraction that encapsulated the structure

of a stigmergic system but provides the high-level abstractions sought by requirement R1 - physical integration.

3.3.8 Security and Privacy

Providing security for pervasive computing environments and ensuring the privacy of the users who use them cannot be addressed directly via the use of stigmergy, though, it does not restrict its inclusion at a later stage. It is possible to argue that since the information in a stigmergic system is kept in the local environment and does not propagate beyond this point that all components and users know of each others existence and so could be considered not to invade on ones privacy. Such features provide a certain level of privacy, however, there may still be some concerns and additional means of securing access to this information may be required to ensure the privacy of users can be maintained. It should also be noted that there are no means of restricting the unauthorised use of devices in a stigmergic system. These type of systems have no means of determining the trustworthiness of users as require by requirement R10 - security and privacy. In conclusion, it can stated that the use of stigmergy on its own cannot address requirement R10 - security and privacy - as described in section 3.1.2.

3.3.9 Summary

It would appear in principle that the concept of stigmergy can be used to address the majority of the requirements stated in section 3.1.2. Table 3.3 provides a summary of the requirements addressed. In the table, a bullet (•) indicates the requirement has been satisfied, while a requirement with no bullet shows that the it cannot be addressed directly through the use of stigmergy. Of the requirements that cannot be directly satisfied via the use of stigmergy, R1 - physical integration - is addressed in chapter 4 where a programming abstraction encapsulated in a high-level scripting language is defined. Requirement R10 - security and privacy - is not directly addressed by the thesis.

Requirement		Supported
R1	Support the physical integration of components into the environment.	
R2	Support the autonomy of components.	•
R3	Support the spontaneous interoperability between arbitrary components.	•
R4	Support the decentralised coordination of component behavior.	•
R5	Provide a scalable solution.	•
R6	Ensure the robust behavior of the system.	•
R7	Support incremental construction and improvement of solutions.	•
R8	Mobility.	•
R9	Adaptability.	•
R10	Security and privacy.	

=no support •=supported

Table 3.3: Summary of requirements addressed.

3.4 A Stigmergic Model for Pervasive Computing

The previous sections have shown, in principle, that the concept of stigmergy can be used to build the type of pervasive computing environments envisioned in section 3.1. Section 3.2.1 started with an outline of the concept and its use in biological systems, while section 3.2.2 showed how the concept has been successfully applied to a number of computer-related problems. The last section took the observations gained from these sections to examine whether a similar approach could be used to build the environments described in section 3.1. It was found that the use of stigmergy in their construction could help address the majority of the requirements necessary to successfully build them. This section looks to continue the work done in previous sections by formulating a model based on the principle of stigmergy that can be used to develop a framework for pervasive computing. The section starts with an overview of the model.

3.4.1 Overview of Model

In modelling a system based on stigmery there are three things that need to be determined, the first is the environment that collections of interacting entities will use to coordinate their

behavior, secondly, are the entities that will use the environment, and thirdly, the means for the individual entities to sense this environment, determine how they react to it, and manipulate it.

In this case it is proposed to use the general principles of stigmergy to create a model for pervasive computing where context information from environmental sensors provides the common environment for the indirect communication between entities. The social insects observed by Grassé [58] are represented as entities in the model. An entity is a person, place, or object as defined by Dey [42]. Entities roam across the environment and act on it by changing their behavior to modify the local environment. The changes in the environment are subsequently reflected in the context information derived from the environmental sensors. Coordinated behavior arises from entities observing their local environment and reacting to the resulting context information according to some rules.

The following sections provide a more detailed description of the model. The next section starts by describing the environment and type of stimulus it provides to entities. The section 3.4.3 continues by providing a more detailed description of how the entities involved are modelled. Section 3.4.4 describes how these entities sense their local environment. Section 3.4.5 outlines the process each of these entities uses to regulate their behavior in the pervasive computing environment, while section 3.4.6 details how the environment is physically changed by the entities.

3.4.2 The Environment

Whether artificial or biological a system using stigmergy is structured around a collection of autonomous entities and the environment they use to mediate their communication. In colonies of social insects, such as termites, the environment is represented by pellets of soil laced with pheromones that have been placed on a growing mound. The resulting stimulus provides the colony with a way of constructing a termite mound. For the nest building seen in social wasps the environment is the structure of the nest itself. For the experiment performed by Holland et al. [71], where simple robots are used to push frisbees into clusters, the environment is the collection of frisbees that the robots are moving around. In the classical

trail-laying and trail-following behavior observed in some species of ants the environment is the pheromones deposited by the ants into the physical environment. In Antnet [27], where stigmergy is used to route packets through a communication network, the environment is the network and each of the nodes on it. To apply stigmergy to pervasive computing it is evident that it is necessary to first determine the environment entities will use to coordinate their behavior.

Section 3.2.1 identified two main forms of stigmergy - quantitative and qualitative. In quantitative stigmergy the stimulus varies in a quantitative manner. Qualitative stigmergy differs from quantitative stigmergy in that it is based on a discrete set of stimuli. Quantitative stigmergy is typically achieved in nature through the deposits of pheromones into the environment, while qualitative stigmergy is done through the physical manipulation of the environment as with the nest building of social wasps. To build a pervasive computing environment with quantitative stigmergy the components would need to able to deposit artificial pheromones into the environment. This could possibility be achieved in either of two ways. Entities could physically deposit substitute pheromones, such as RFID tags, into the physical environment as Mamei et al. [93] has attempted to do. These deposits could then be read by other entities thus having an effect on their subsequent behavior. However, such an approach would most likely be prohibitively expensive and would leave an unwanted residue in the physical environment. The alternative would be to provide entities with a means of depositing artificial pheromones in a virtual environment. Building such an environment for the type of pervasive computing environments discussed in section 3.1.1 could prove to be extremely challenging. Entities would need to be able to maintain the state of the environment so that a consistent view could be provided to all entities in the local environment. To achieve this in a system that is highly decentralised and where entities interact on an ad-hoc basis is a particularly difficult problem. It is also confounded by the nature of the information being stored which is typically tied to the geographical location in which it is deposited. To maintain the information at those locations requires the presence of a computational node, however, for the type of environments envisaged this may not be possible with the result that information may be lost.

Qualitative stigmergy may provide a more suitable approach for pervasive computing. In this type of stigmergy entities change their behavior in response to qualitatively different stimuli. Typically, this is achieved through entities physically changing the environment in some way and other entities reacting to the alteration. Used in pervasive computing the environment would translate into the physical space occupied by users and other everyday objects. Through the use of actuators entities would be able to physically manipulate this environment. For example, an entity could turn on a light, display information on a screen, or even alter what is playing on a jukebox. Other entities, using sensors, would be able to sense these alterations and react to them. Each of these entities would receive qualitatively different stimuli from the changes made to the environment by other entities so triggering different behavior from them. It must also be noted that this type of environment can always be changed locally by entities and the changes made to it are more likely to persist in the environment long enough to effect the subsequent behavior of other entities. These characteristics allow such an environment to meet the minimum requirements, set out by Holland et al. [71] and highlighted in section 3.2.1, to support stigmergy.

Instead of directly using the raw data from sensors it is proposed to use context information derived from sensors to provide the common environment for the indirect communication used by entities. Context information is any information that can be used to characterise a situation [42]. It is typically used in pervasive computing [61, 134, 78, 64, 56] to adapt and tailor the behavior of a system to meet the requirements of a user. However, in this case it is used to describe the local environment for an entity. It provides a much higher level of representation of the physical environment than would otherwise be possible from viewing the raw sensor data. A combination of sensor fusion techniques are typically used to fuse the sensor data together to provide this representation. The process also helps to remove the ambiguity normally associated with interpreting sensor readings from the environment.

To build up this representation entities use the sensors in their vicinity to sense the state of the physical environment. The sensors would typically measure different aspects of the environment local to them. The data retrieved from these sensors allow entities to derive the context information that they subsequently use to coordinate their activities in the

environment. Where the sensors are placed and how they are configured in relation to the entities is discussed later in section 3.4.4.

3.4.3 Entities

It seems logical to represent the social insects observed by Grassé [58] as entities in a pervasive computing system, where an entity is a person, place, or object. This choice of entity is in most part influenced by Dey's definition [42] of an entity. Though, it should also be noted that a person, place, or object are either physically or computationally the active entities in a pervasive computing environment and it is these entities which form the bulk of interactions in the environment. So by defining an entity in this way we look to create a large collection of interacting entities that can coordinate their behavior through the use of stigmergy. The environment, as described above in section 3.4.2, provides entities with the medium to coordinate their behavior. For these entities to use this environment, and to support stigmergy, they need to satisfy the requirements set out by Holland et al. [71] and highlighted in section 3.2.1 - they must firstly be able move through the environment, and secondly, be able to act on it.

Some entities will be highly mobile while others may move irregularly or not at all. However, when viewed together in the context of the type of pervasive computing environments being developed it can be expected that enough activity can be generated to support a stigmergic system. The mode used by these entities to move through the environment may vary depending on the entity. It can be assumed that a large portion will be self-propelled, in that, they are able to move through the environment under their own means. People, autonomous robots, or vehicles are good examples of such entities. Other groups may exploit these entities to roam across the environment by attaching to them or being carried by them. Such an example could be a PDA used by a person or a radio in a car. As these entities roam through the environment it is also important that they are able to act on it. This implies that each entity must first be able to observe their local environment, and secondly, be able to change it. The previous section outlined how this can be achieved in a pervasive computing environment. Sensors allow entities to determine the context information of the local envi-

ronment and through the use of actuators they are able to manipulate it according to a set of conditions. Under this mapping the chosen entities are capable of satisfying the minimal requirements, set by Holland et al. [71], to support stigmergy. To determine how each of these entities are modelled in a pervasive computing environment it is first necessary to understand how individual entities adapt their behavior to changes in the local environment.

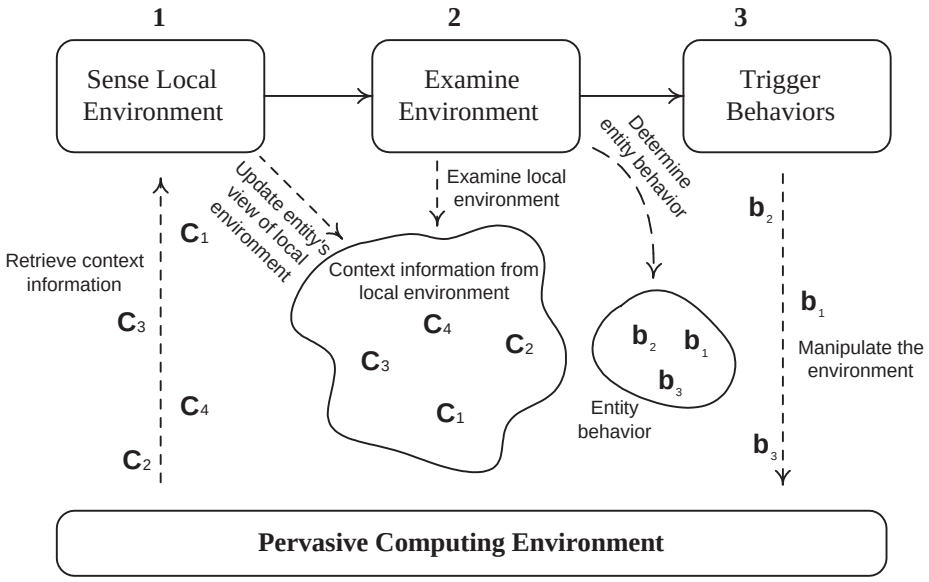


Figure 3.1: Stigmergic Model.

Figure 3.1 provides an overview of this process. It shows how each entity in the system can change its behavior in response to qualitatively different stimuli. The first step is to examine the local environment of the entity and to provide it with a representation it can use to adapt its behavior. This requires the entity to sense its local environment and to retrieve any context information that may be of use to it. How this information is retrieved is discussed in greater detail in section 3.4.4. The next stage dictates how each entity responds to the different stimuli found in the local environment. It first examines the context information of the local environment to determine whether the right stimuli are present to trigger any of the entity's behaviors. The process maps the entity's local environment onto the set of behaviors the entity can exhibit. Typically, it follows a predefined set of rules that allow the

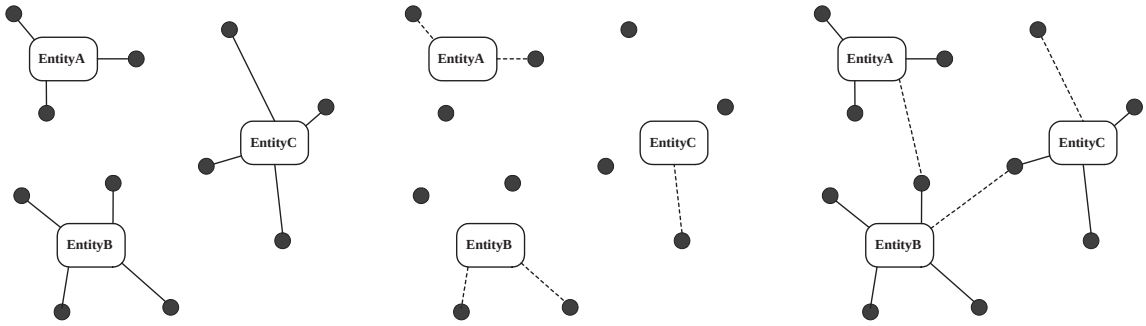
entity to react to the local environment. More detail is provided in section 3.4.5. The last stage of this process involves executing the specific behaviors that have been triggered. Each of these behaviors defines how the entity reacts to a particular stimulus. The actuators used to implement the behavior allow the entity to physically change the local environment. Such alterations in the environment effect the subsequent behavior of the other entities, thereby initiating a stigmergic response. Section 3.4.6 describes in greater detail the mechanisms that allow entities to change the environment.

3.4.4 Sensing the Environment

In biological systems such as social insects each individual in the colony has an in-built ability to sense the environment around them. Using sight, smell, or touch they are able to detect relevant changes in the environment that allow them to coordinate their actions using stigmergy. In pervasive computing sensors provide the technology that allow computational devices and applications to sense the physical environment around them. For the entities, described in section 3.4.3, to sense their local environment they need to use sensors. In biological systems each individual uses a fixed collection of senses to determine the environment. These are permanently associated with each individual. The equivalent configuration for entities in a pervasive computing system is to use a set of sensors directly connected to each entity. Figure 3.2(a) provides an illustration of such a configuration. However, for pervasive computing their are other possible configurations that should be considered and which may provide a better solution.

Figure 3.2 illustrates three possible sensor configurations that could potentially be used in such a system. The first, as described above, permanently associates a collection of sensors with each entity. The sensors are not shared between entities and the entity only uses those sensors in its collection to determine the context information of the local environment. Typically, these sensors would be physically connected to the entity in some way and would move with the entity as it roams through the environment. Such a configuration would help satisfy requirement R2 - autonomy of components - as each entity would be self-contained in its ability to sense the local environment. However, for pervasive computing detecting the

Each of the black dots represents a sensor in the environment. A solid line connecting the sensor to the entity indicates a permanent association between the two, while a dotted line shows the sensor is only temporally assoicated with the entity.



(a) Each entity is permanently connect to a collection of sensors.

(b) Sensors embedded into the environment. Entities discover those sensors they wish to use and temporially connect to them.

(c) Entities are permanently connected with a number of sensors and termporally assoicated with other sensors they wish to use.

Figure 3.2: Possible Sensor Configurations.

state of the environment is a complicated task due to the unreliability of sensors and the difficulty of interpreting the data correctly. Therefore, such an approach may not provide the best results as the limited set of sensors might not be able to provide a reliable picture of the local environment. Entities would have to carry a large selection of sensors to be confident in sensing their local environment correctly. While entities with large resources may not have a problem, others, running on smaller resource constrained devices may have.

An alternative is the sensor configuration shown in figure 3.2(b). In this case sensors are embedded into the physical environment and are not directly associated with any particular entity but shared between them all. To sense the environment entities retrieve data from sensors in their vicinity by temporally connecting to them. As they move through the environment they continue to select sensors that provide them with data on the local environment. To be able to dynamically discover and use the data from the sensors in this way entities would require a certain level of knowledge on the of types sensors that they wish to use. A type of metadata that would describe the location of the sensor, its coverage, data provided by the sensor, accuracy of the data, and frequency of the data produced by the sensor would most likely be required by the entity. This would allow entities choose the right sensors to use in their vicinity and ensure they are able to interpret the data received from the sensor.

The use of such an approach would provide entities with a greater selection of sensors as they are embedded into the environment and accessible to all entities. This should give entities access to a greater number of sensors with which to sense their local environment, while also eliminating the need for entities to maintain sensors so reducing the burden on each entity. However, it does increase the complexity as entities would be required to discover the sensors as they move through the environment. They would also have to retrieve the data from the sensors. Furthermore, the autonomy of entities would also be reduced as entities would require other parts of the pervasive computing environment to provide them with information on the local environment. A more serious issue for this approach is it requires the pre-installation of the sensors into the environment before it can be used. This is contrary to the type of pervasive computing environments envisioned in section 3.1 where the environments are intended to evolve in an ad-hoc fashion, emerging at hotspots of activity where users require and want to use them. This implies that the location of the interactive environment is not know beforehand, making the pre-installation of the sensors into the environment particularly difficult to achieve.

By combining some of key concepts from the first approach and those from the last it may be possible to obtain a better solution. Figure 3.2(c) provides an illustration of such an approach. In this configuration entities are permanently associated with a selection of sensors that allow them to sense the basic parameters of their environment. As they move through the environment they can also select and use the data from sensors connected to other entities and those embedded within the environment. The same method of discovering and connecting to the sensors is used as in the previous approach. By sharing sensors at this level entities can cooperate in obtaining a better understanding of their local environment without the burden of having to maintain a large collection of sensors. In this way they can gain access to a wide selection of sensors while still maintaining their autonomy in the environment. The approach also avoids the need to pre-install sensors into the environment as they can be embedded in each entity.

The environmental sensors provide entities with a means of determining the context information of the local environment. It is the context information that is used as the common

medium for the indirect communication between entities and not the raw sensor data. Using the above sensor configuration entities can apply different sensor fusion techniques to fuse the data received from sensors. This allows entities to determine the context information of the local environment and remove the ambiguity typically associated with sensor readings. In practice, entities can use one of two approaches to achieve this. They can either accomplish it on their own by taking all the available sensor data from the environment to capture the context information of the local environment. This has the potential to be quite expensive and resource intensive for the entity. The other option is to allow entities cooperate in determining the context information of the local environment. In this way entities derive context information for different parts of the environment and share it with those in its vicinity. Together the entities can build up an entire picture of the environment. Such an approach would reduce the cost of sensing the environment while still allowing entities to maintain their autonomy within the environment.

While the proposed approach to sensing the local environment does not strictly follow that used by individuals in a biological system using stigmergy it is felt that by providing a such cooperative approach to sensing the environment entities would have a better chance of retrieving the context information from the local environment in a timely manner. It should also be noted that the approach satisfies the requirements outlined in section 3.1.2.

3.4.5 Interpreting the Local Environment

In this model entities change their behavior in response the qualitatively different stimuli. The context information derived from the sensors provides entities with the means of describing these discrete changes in their local environment. Entities use the stimulus that this provides to adapt their behavior. How an entity's behavior changes depends on the type of stimuli that effects their behavior and on the behaviors the entity can perform in the pervasive computing environment. It can be expected in these types of environments that entities will have a finite set of behaviors they can use to change the state of the environment. For example, a door can either open or close, or a jukebox can play music, pause, or stop playing. Each of these behaviors would utilise the different actuators embedded in the environment to physically

alter it. To determine how individual entities respond to changes in their local environment it is necessary for them to examine the context information for the presence of any stimuli that might change their behavior. If a discrete stimulus is found in the environment the behavior relating to can be triggered, thereby changing the physical environment for other entities and allowing them to react to it.

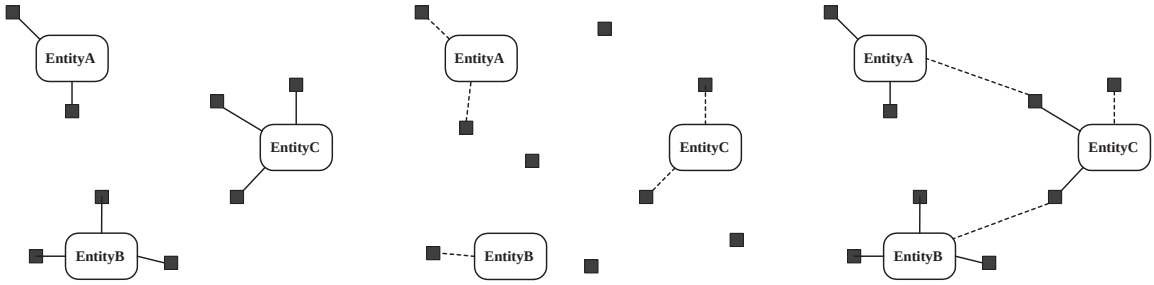
The method of mapping an entity's current view of the environment onto the behaviors they exhibit is a matter that needs further consideration. Each entity in this model has the ability to respond to their own discrete set of stimuli types. What might effect the behavior of one entity may effect another in a different manner or not at all. To achieve this each entity has to be able to examine their local environment, identify the stimuli that effect their behavior, and map that onto the behaviors which will eventually change the environment. A rule-based approach would appear to be the most appropriate means of accomplishing this. Such an approach has been found to be quite effective in a number other systems [96, 48, 71] using stigmergy. It typically uses a minimal amount of computation and requires little or no internal state to run. This makes it especially suited to pervasive computing where resource constrained devices are the norm. Furthermore, the minimal amount of computation involved means the system can act in a timely manner and allows it to respond quickly to a rapidly changing environment.

Defining the individual behavior of each entity is achieved through specifying a set of rules that map stimuli found in the local environment onto the behaviors exhibited by each entity. Programming of these rules are done via a scripting language described in chapter 4. The scripting language defines a discrete set of stimuli that influence the entity's behavior in terms of the context information observed in the local environment. The resulting contextual predicates allow the scripting language to define a set of rules for mapping the entity's local environment onto the behaviors it can perform in the pervasive computing environment.

3.4.6 Manipulating the Environment

The indirect communication used by social insects to interact is mediated through modifications of the local environment. This is achieved in either one of two ways depending on the

Each of the small black squares represents an acquator in the environment. A solid line connecting the acquator to the entity indicates a permanent connection between the two, while a dotted line shows the entity is only temporally assoicated with the actuator.



(a) Each entity is permanently connected to a collection of actuators.

(b) Actuators are embedded into the environment. Entities discover those actuators they wish to use and temporially connected to them.

(c) Entities are permanently connected to a number of actuators and termporally connected to other actuators they wish to use.

Figure 3.3: Possible Configurations for Actuators.

type of stigmergy used. For quantitative stigmergy deposits of pheromones are made into the environment. In qualitative stigmergy individuals typically physically change the environment in some way. The nest building of social wasps is a prime example of such behavior. In this model entities use qualitative stigmergy to coordinate their actions within a pervasive computing environment. On sensing a specific stimulus entities trigger the particular behavior relating to it. The implementation of these behaviors uses the different actuators in the environment to perform the behavior. These allow the entity to physically manipulate the state of the pervasive computing environment. Section 3.4.2 has already described how the use of actuators allows entities to modify such an environment.

As with the sensor configurations, discussed in section 3.4.4, there are also a number of possible configurations that can be used for actuators in a stigmergic model for pervasive computing. Figure 3.3 provides an illustration of these configurations. The first, permanently connects a series of actuators to each entity. The actuators are not shared between entities and are only used by the entity to perform specific behaviors in the environment. The configuration is basically equivalent to the one used by social insects. An alternative to that approach is the one shown in figure 3.3(b) in which the actuators are embedded into the environment and are not directly associated with any particular entity but shared between them all. As

entities move through the environment they dynamically discovery the actuators they require to implement a specific behavior. A similar mechanism to that used to locate sensors in section 3.4.4 is required to discover actuators in the environment. The last option combines the concepts from the first two approaches. Figure 3.3(c) provides an illustration of this approach. In this configuration entities are permanently connected to a number of actuators. As they move through the environment they can also discover actuators connected to other entities and those embedded within the environement and use them in combination with the ones they have to perform different behaviors.

The first option ensures that all the actuators required to perform a behavior are always available for the entity to use. This should make it possible for the entity to act in a more timely manner as there is no need to discover or establish a connection to the actuator, in doing so, allowing the entity to respond quickly to a rapidly changing environment. Furthermore, such a configuration promotes the autonomy of the entity as it is self-contained in its ability to modify the local environment.

The second configuration, shown in figure 3.4.4, provides a good solution for an interactive environment where the location of the environment has been predetermined. This type of configuration requires the per-installation of the actuators into the environment before it can be used by entities. The advantage of the approach is that is allows entities to reuse actuators in the environment and so reduce the need for duplicate components in the system. In combination with the sensor configuration, shown in figure 3.2(b), it could provide a very good solution where both the sensors and actuators are embedded in the environment. The entities would roam across the environment choosing the sensors and actuators they wish to use to implement their stigmergic behavior. Unfortunately, the type of pervasive computing environments envisioned in section 3.1 evolve in an ad-hoc fashion, implying, that the location of the interactive environment is not know beforehand. This makes it particularly difficult to pre-install the actuators in the location prior to the entities using it. Therefore, such a configuration is not a viable option for this model.

The last configuration in some way provides a solution to this problem by directly connecting the actuators to the entities and allowing entities to share them as well as the actuators

embedded within the environment between themselves. However, it is not clear that the additional access to actuators gives any extra benefit to the entity other than increasing the complexity of using them. In this case the model allows entities to have a finite set of behaviors they can use to alter the state of the environment. It is arguable that access to an increased selection of actuators would facilitate the improved implementation of the behaviors for the entity. There is also the prospect that entities won't be able to find the actuators they require to implement a specific behavior or there is an increased period of time required to discover and connect to an actuator. Such issues affect the entities ability to respond to a rapidly changing environment. This is also a problem that affects the second approach.

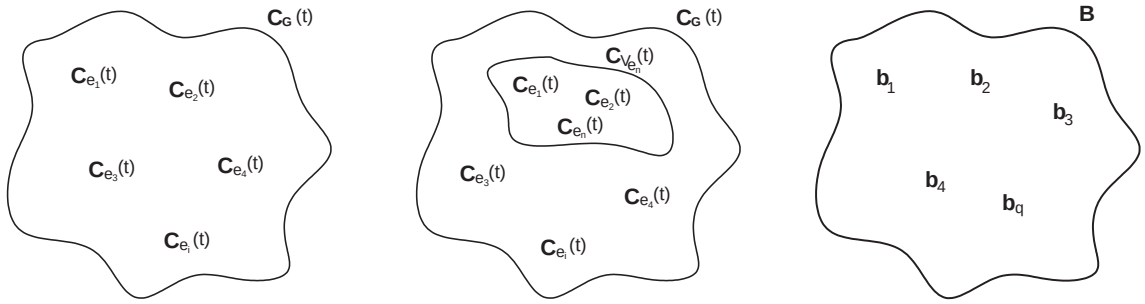
It would appear that the first configuration, shown in figure 3.3(a), is the best approach for the type of pervasive computing environment being built. In this case all the behaviors for the entity can be implemented by actuators directly connected to the entity. The approach ensures entities can always react efficiently to the environment without the need of additional support.

3.4.7 Summary

The model presented in this section provides a highly decentralised method of managing the behavior of entities in a pervasive computing environment. It allows individual entities to change their behavior in response to a discrete set of stimuli. Over time coordinated behavior can emerge as different entities change their behavior in response to those changes made by other entities to the environment. Context information from environmental sensors provides the common medium for this indirect communication. It describes the physical state of a pervasive computing environment and any changes made to it are reflected in the context information derived from it. In this model entities share the cost of sensing the environment through their cooperation in the sharing of sensor data and the context information extracted from it. Entities modify the local environment through the use of actuators directly connected to each entity. The individual behavior of an entity is determined by a set of rules. Each of these rules map a particular stimulus onto a set of behaviors an entity can perform in the pervasive computing environment.

3.5 Key Features of Model

The last section presented the basis of a model founded on the concept of stigmergy that can be used to develop a framework for pervasive computing. This section provides a summary of that model in terms of a formal representation that may be used in implementing the framework. It starts by first defining what the local environment of an entity is. It continues by defining the behaviors entities use to change the state of the environment and the process used to respond to different stimuli found in the local environment. The following sections outline the process in more detail.



(a) Global context (b) Contextual view (c) Behavioral set

Figure 3.4: Using Stigmergy in an Pervasive Computing Environment

3.5.1 The Local Environment

The context of an entity e_i in the environment at time t is represented by $C_{e_i}(t)$. It is the context information used to describe the situation of the entity. Equation 3.1 defines $E(t)$, the set of all entities that exist at time t.

$$E(t) = \{e : e \text{ is an entity} \wedge e \text{ exists at time } t\} \tag{3.1}$$

Figure 3.4(a) represents the context of every entity in the pervasive computing environment at a particular time. It is the global context $C_G(t)$ of the environment and is defined by equation 3.2. The global context describes the state of the whole environment at a particular instance in time.

$$C_G(t) = \{C_{e_i}(t) : e_i \epsilon E(t)\} \tag{3.2}$$

All information contained in $C_G(t)$ is not required by each individual entity, as the behavior of an entity is only dictated by the context of its local environment. Figure 3.4(b) illustrates a subset of the context information relevant to an entity. It represents the local environment and defines the entity's *contextual view* $C_{V_{e_n}}(t)$, as defined in equation 3.3. It holds all context information in $C_G(t)$ that is relevant to the situation of entity e_n at time t. An entity's context $C_{e_i}(t)$ is included in entity e_n's contextual view if the entity is within a certain proximity. The notion of proximity is used to define what is local to the entity. This is captured in equation 3.3 where the function $L(e_i, e_n)$ is used to determine proximity and returns *true* if entity e_i is within the required proximity of entity e_n.

$$C_{V_{e_n}}(t) = \{C_{e_i}(t) : C_{e_i}(t) \epsilon C_G(t) \wedge L(e_i, e_n) = true\} \tag{3.3}$$

3.5.2 Defining Entity Behavior

The behavioral set B, shown in figure 3.4(c) and defined in equation 3.4, represents a finite set of behaviors that entities can use to change their local environment. For example, a light can either turn itself on or off, or a jukebox can play music, pause, or stop playing. The behavioral set defines how entities can change their behavior to modify the environment.

$$B = \{b : b \text{ is a behavior of an entity}\} \tag{3.4}$$

3.5.3 Reacting to the Local Environment

The last stage manages how each individual entity adapts its behavior to reflect changes in the local environment. Equation 3.6 defines the function M for mapping $C_{V_{e_n}}(t)$ onto $\mathcal{P}(B)$[1]. $C_{V_e}(t)$, defined in equation 3.5, represents the collection of all contextual views. The function maps the entity's context information from the local environment onto a behavior, thus initiating a stigmergic response to the environment. For example, if set B is defined by

[1] The power set of behavioral set B.

figure 3.4(c) and $C_{V_{e_2}}(t)$ is the contextual view of e_2 at time t then function M could possibly map $C_{V_{e_2}}(t)$ onto $\mathcal{P}(B)$ as follows $M(C_{V_{e_2}}(t)) = \{b_3\}$.

$$C_{V_e}(t) = \{C_{V_{e_i}}(t) : e_i \epsilon E(t)\} \tag{3.5}$$

$$M : C_{V_e} \rightarrow \mathcal{P}(B) \tag{3.6}$$

The proximity function L, the behavioral set B, and the M function provide three primitives that define how individual entities behave in response to changes in the local context state of the environment. Over time system-level behaviors may emerge as different entities change their behavior in response to the changing state of their local environment.

3.6 Summary

The chapter has argued that the use of swarm intelligence techniques, such as stigmergy, can help in the construction of pervasive computing environments. Section 3.1 used the observations made in chapter 2 to sketch an outline of the next generation of pervasive computing environments that addresses some of the short comings of the environments presented in chapter 2. The following analysis establishes a set of requirements for developing such pervasive computing environments and determined the extent to which the state of the art supports them. The chapter then proceeded in section 3.2.1 to outline the phenomena of stigmergy while section 3.2.2 reviewed its use in other areas of computer science research. Section 3.3 examined how stigmergy could be used to build pervasive computing environments and indicated how it could be used to address the requirements stated in section 3.1.2. Section 3.4 developed the understanding gained from section 3.3 to present a model based on the concept of stigmergy that can be used to develop a framework for pervasive computing, while the last section provided a more formal representation of this model that can be used in implementing the framework.

The aim of the chapter has been to develop a highly decentralised approach to organising components of a pervasive computing environment that supports the spontaneous interac-

tion between entities and provides robust system-wide behavior. The use of stigmergy has provided a means of achieving this. It has enabled an approach, encapsulated in the three primitives presented in section 3.5, for developing pervasive computing environments. While the proposed approach has similarities to software agents, in that an entity can be thought of as an agent, it differs from conventional agents in the approach used to communicate and coordinate entity behavior.

Of the requirements stated in section 3.1.2 only one cannot be fully supported through the use of the model presented in section 3.4 and 3.5. Requirement R1 - physical integration - looks for to support the physical integration of components into the environment and as stated in section 3.3.9 cannot be supported through the direct use of stigmergy. The next chapter looks to address the situation by providing a high-level scripting language, which will aid in fulfilling requirement R1 - physical integration.

Chapter 4

Defining Entity Behavior

This chapter looks to advance the concepts introduced in the previous chapter to develop a programming model, encapsulated in a high-level scripting language, to ease the implementation and deployment of pervasive computing applications. Focusing on defining entity behavior, the chapter describes a scripting language that can be used to specify the behavior of an entity in relation to their surrounding environment. The inspiration for the scripting language is drawn from the stigmergic model developed in chapter 3.

In proposing a scripting language the objective has been to provide those developing pervasive computing environments with a tool that combines expressiveness and simplicity with the ability to abstract the complexities of dealing with the underlying technologies. The aim is to allow developers concentrate their efforts on characterising the behavior of pervasive computing environments rather than low-level system development, and to provide a means of facilitating the incremental construction and improvement of solutions over the lifetime of the environment. In addition, the goal is to develop an approach that provides support for requirement R1 specified in section 3.1.2 - supporting the physical integration of components into the environment. The scripting language provides the high-level abstractions needed to fulfill this requirement.

The use of scripting technologies, as highlighted by Ousterhout [115], allows for the accelerated development of pervasive computing applications, which can evolve rapidly over time. For pervasive computing these characteristics are imperative for the healthy evolution

of the environment. It cannot be assumed that a pervasive computing environment will remain static, it must grow with its participants and change its behavior to suit them. Using a scripting language will also help avoid the *"big messy C program(s)"* that Coen [36] observed when developing pervasive computing environments. It provides the glue to combine components, while separating the complexities of the underlying system from the creative side of characterising the behavior of the entities within the pervasive computing environment. While of use scripting technologies does come at a cost, in that, the interpretation can be slow and in general consume more resources than directly executed code, it is out weighed by the advantages that such an approach provides.

The stigmergic model defined in chapter 3 provides the foundations on which this scripting language is built. In particular, the three primitives defined in section 3.5 - the proximity function L, the behavioral set B, and the M function - form the basis of the language. Each of these primitives are defined in the scripting language and collectively determine how an entity behaves in the pervasive computing environment. The following sections outline in more detail the approach taken after a review of current scripting technologies used in pervasive computing is provided.

4.1 Current Scripting Technologies

Scripting technologies have been used in pervasive computing to both define the behavior of components and to specify how those components are composed to build pervasive computing environments. Their use in these roles has helped pervasive computing systems to abstract the complexities of the underlying system, and aided the rapid development of applications for these types of environments.

An example of a project that has used scripting technologies to define the behavior of components is the TEA (Technology for Enable Awareness) [142] project. TEA uses a scripting mechanism to describe basic actions to be performed by an application. The actions can be performed when entering a context, when leaving a context, and while in a certain context. Their framework concentrates on adapting the behavior of small devices such as mobile phones.

Brown et al. [18, 17] use scripting technologies for a similar purpose. In their stick-e system

they use the Standard Generic Markup Language (SGML) to describe the context of when a note is to be triggered. The approach provides an expressive means, which is extensible, of defining a full range of contexts in which to trigger a note. Schmidt [140] extends this approach by adding two further concepts - additional trigger attributes and the ability to bundle context attributes into groups. The trigger attributes allow the triggering of actions in a similar way to that of the TEA project in that actions can be performed when entering a context, when leaving a context, and while in a particular context. The grouping of contexts allows the system to trigger actions when one, all, or none of the grouped context attributes are matched. The approach provides a more expressive means of describing how applications can change their behavior.

Pinhanez defined an interval scripting language [125] to allow applications describe the temporal behavior of components. Based on PNF-networks [122] Pinhanez et al. have used the interval scripting language to create interactive environments such as SingSong [124] and It/I [123]. These are story-based interactive systems that script the interactions between users and the environment with the interval scripting language. The interval script describes the temporal relationships between different states of the environment, defining when to stop and to start other actions. Pinhanez's PNF-networks are based on Allen [5] temporal intervals.

RCSM (Reconfigurable Context-Sensitive Middleware) [175] is another project that uses scripting technologies to control the behavior of components. RCSM is an object-based framework that uses an IDL-based language called CA-IDL to generate context-sensitive objects. These objects run on a customised ORB that supports the communication and context-awareness between objects. Developers use the CA-IDL language to define context variables that are used in temporal expressions within the script to trigger either local or remote method invocations on objects in the pervasive computing environment.

Gaia [133] uses high-level scripting languages in a different manner to the above, in that, it uses scripts to compose the components of a pervasive computing system into applications that can be used by those in the environment. As already discussed in section 2.3.7, Gaia has an application framework based on the Model-View-Controller [88] user interface paradigm. They use scripts, based on the interpreted language Lua [74], to describe how to combine the

various components in the environment to form an application.

The scripting language described in this chapter applies some of the techniques used in the above projects to define entity behavior and to facilitate the emergence of pervasive computing environments from collections of autonomous entities. In particular, the scripting language uses similar methods to RCSM [175] in declaring context and triggering actions but extends the approach to include the full range of temporal relationships used by Pinhanez [125] and defined by Allen [5]. However, the approach also includes a number of novel techniques for extending entity behavior and for facilitating the incremental development of pervasive computing environments. The approach also includes methods for tailoring the behavior of entities by the passing of context to the actions.

4.2 A Scripting Language for Defining Entity Behavior

In the stigmergic model defined in chapter 3 entities roam across the environment and act on it by changing their behavior to modify the local environment. The changes in the environment are subsequently reflected in the context information derived from the environmental sensors. Coordinated behavior arises from entities observing their local environment and reacting to the resulting context information according to some rules. It is proposed to use a scripting language to specify these rules and in the process define the behavior of the entity. Contextual predicates specify the stimuli that allow the scripting language to coordinate the behavior of an entity with the state of the local environment. Even though there are no scripting primitives providing direct communication between entities, coordinated behavior can still be programmed through the principle of stigmergy.

Using such an approach provides a highly decentralised method of managing the behavior of entities in a pervasive computing environment. Individual entities can be programmed to change their behavior to react to specific contextual stimuli. Over time coordinated behavior can emerge as different entities change their behaviors in response to the changing behaviors of other entities. What is important to note is that each entity is developed individually to react to the stimuli that is of interest to them. This allows the incremental development of pervasive computing environments as new entities can be added and old ones upgraded at

any point in the lifetime of the environment. In doing so, the scripting language aids in the fulfilment of requirement R7 - supporting the incremental construction and improvement of solutions - defined in section 3.1.2. The approach also provides a means of abstracting the complexities of dealing with the underlying technologies, allowing those developing pervasive computing environments to concentrate on the creative side of defining entity behavior. In taking this approach the scripting language provides the high-level abstractions sought by requirement R1 - supporting the physical integration of components into the environment - defined in section 3.1.2.

In proposing a new scripting language the objective has been to provide a tool to simplify the development of pervasive computing environments. Its role in such development is to support developers in defining entity behavior. The high-level abstractions are expected to facilitate those less experienced in developing such environments to specify entity behavior. The system level development, such as the integration of sensors, is anticipated to be handled by those with greater knowledge of the area. While there is an overhead in defining such a domain specific language, compared to using or extending an existing language, the advantaged gained in providing the right level of abstraction out weights the disadvantage of having to specify such a language. It should, however, be noted that a base language, in this case Java, is used in combination with the scripting language for the system level development of the environment. A more indept discussion of the development process and of the system level development is provided in chapter 5.

The foundations of the scripting language are built upon the stigmergic model, in particular the three primitives defined in section 3.5 - L, B, and M - form the basis of the language. L, the proximity function defines an area over which an entity has influence. B, the behavioral set represents the set of possible behaviors an entity can use to manipulate the local environment. The M function provides the means of mapping an entity's local environment onto the behavioral set, B. Together, they allow the scripting language to define how an entity is to behave and how it is to coordinate its activities within a pervasive computing environment. The following sections provide a more detailed description of the primitives used in the language to define entity behavior.

4.3 Basic Structure

This section describes the main structure and capabilities of the scripting language. Section 4.3.1 starts with an overview of the language describing the basic structure and usage. Section 4.3.2 describes how L, the proximity function, is defined in the scripting language. Section 4.3.3 outlines how B , the behavioral set, is specified for an entity. Section 4.3.4 describes how the M function is defined. A full description of the grammar of the language can also be found in appendix A.

4.3.1 Overview

The scripting language uses an interpreter that takes a text file containing a description of the desired behaviors and translates them into intermediate objects that the framework uses to represent the behavior of individual entities. Details of the methods used to interpret the script and to translate it into intermediate form are described later in chapter 5. The behaviors described in the text file characterise how a particular type of entity behaves in an environment and can be reused for all entities of that type.

```
1 desklight extends light{
2   ...
3 }
```

Listing 4.1: Declaring a script.

For instance, the code shown in listing 4.1 defines a script for an entity of type *desklight*. Any entity that can be categorised as a desklight may use the behaviors described in the script to regulate how they behave. The script defines these behaviors in terms of the three primitives - L, B, and M - outlined in the stigmergic model, and which are described in more detail in the coming sections. By using a script to characterise the behaviors of a particular type of entity and not for a specific entity it makes it possible to reuse the script for all entities of that type in a pervasive computing environment. Taking this approach makes the development of entities easier and also promotes the reuse of code. However, there is a need

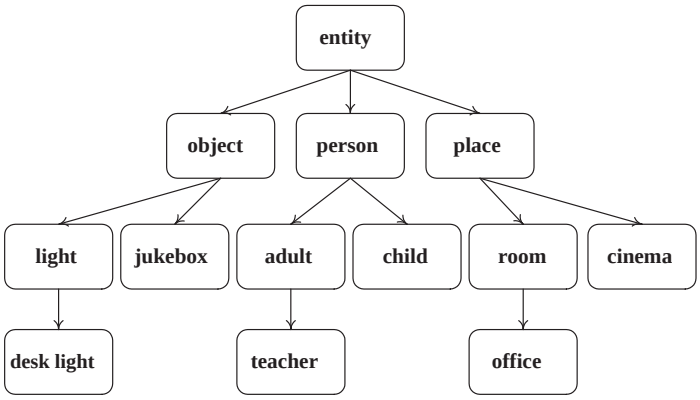


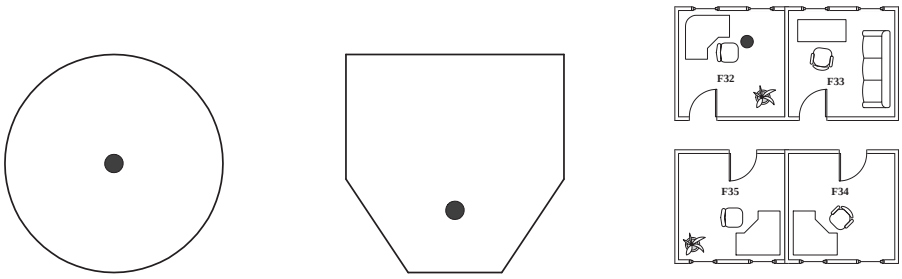
Figure 4.1: Script inheritance hierarchy.

to be able to tailor the behaviors defined in the script for different sub-types of entity. For example, a desklight can be categorised as a light but may behave in a subtly different manner. To manage these aspects the scripting language allows a script to inherit from another script by extending a preexisting script. This allows the script to inherit the behaviors from a base script and from there it can adjust the inherited behaviors to meet the requirements for that sub-type of entity. In the example shown in listing 4.1, the *desklight* inherits from *light*. The use of such an approach helps to promote the reuse of code but also aids in rapid development and incremental construction of pervasive computing environments, in that, it is possible to reuse existing functionality and to be able to extend it. The semantics of inheritance are described in later sections.

As can be expected with inheritance relationships a tree-like hierarchical structure is formed. The inheritance hierarchy for this language, see figure 4.1, is influenced by the presence of four predefined scripts - *entity*, *object*, *person*, *place* - that enforce a structure on the hierarchy. The latter three - *object*, *person*, and *place* - represent the entities in the stigmergic model with *entity* being the base script for these. The reason for imposing such a structure is to match the type of entities defined in the stigmergic model and to ensure that all entities developed with the scripting language are of that type. In the example shown in listing 4.1, the *desklight* inherits from *light* but what can also be seen from figure 4.1 is that

light inherits from *object* which inherits from *entity.* While desklight script inherits behaviors from light, it also indicates that it is a type of object and an entity.

The following sections describe the structures used to define entity behavior, in so doing, focuses on how the three primitives - L, B and M - are defined in a script and on the semantics used for inheritance from a script.



(a) Radius (b) Polygon (c) Symbolic

Figure 4.2: Proximity

4.3.2 Proximity Function

In the scripting language, L, the proximity function can be defined as either a radius, polygon, or symbolic area around an entity. Any context emanating from this region will be inserted into the entity's contextual view as has already been described in section 3.5. In the first example shown in listing 4.2 the proximity is set to be a 5 meter radius around the current entity. This example can be seen in figure 4.2(a).The next example uses pairs of coordinates to define a polygon: the unit of measurement is meters, and the reference point for the polygon is the position of the entity. The polygon defined by the sample code is illustrated in figure 4.2(b).

```
1 proximity(5) //cirle
2 proximity(-5,-5,-10,5,-10,20,10,20,10,5,5,-5) //polygon
3 proximity("F32") //symbolic location
```

Listing 4.2: Code for proximity functions.

Figure 4.2(c) shows the use of symbolic proximity, where a predefined area can be used

to specify the proximity around an entity. This type of proximity is useful when there is a strong definable boundary, such as a room or building. It helps filter out interference from entities which are nearby, but are not relevant to the current situation, i.e. are outside the boundary. The last example shown in listing 4.2 illustrates how this kind of proximity can be defined in the script, where it is set to an office called *F32*. Typically, a symbolic proximity is mapped to an absolute location, or relative location which is either specified beforehand or learnt during the lifetime of the entity. How this is achieved is described in more detail in chapter 5.

The proximity function, L, must be defined for each entity so that it can determine the scope of its local environment. This is typically achieved in the scripting language by calling the proximity function at the beginning of the script to initialise L for the entity. However, it may also be inherited from a base script if L is not defined in the extended script. A sub script can also, if the current definition of L in the base script is not compatible, redefine L by calling the proximity function in the script again. In essence, the proximity function, L, has to be defined in the script or in one of its base scripts, in which case the definition used for L is the last inherited one.

4.3.3 Behavioral Set

B, the *behavioral set,* defines the set of possible behaviors that can be performed by the entity. The actual implementation of a behavior is not provided in the script but in Java following a particular API defined by the framework. The specification of the API is covered in section 5.2.4.3. The script specifies behaviors that a particular type of entity can perform. It achieves this by declaring a behavior with a corresponding reference to the implementation of that behavior. For example, in listing 4.3 behaviors *on* and *off* are declared with reference to the classes that implement the behavior for the entity. In this case, it is the behaviors for turning a light on and off. When the behavior is invoked it executes the Java code that defines the specific behavior that allows the entity to manipulate the environment.

The declaration of a behavior for an entity can also be inherited from a base script. For instance, the desklight from listing 4.1 could have inherited the *on* and *off* behaviors from

```
1 behavior on = "ie.tcd.cs.lighton"
2 behavior off = "ie.tcd.cs.lightoff"
```

Listing 4.3: Declaring behaviors.

light. It could then use them along with the behaviors it declares to define how desklight entities are to behave. It is also possible, if a particular inherited behavior does not suit, to reassign the behavior to use a different implementation. This may be required if a sub-type of an entity uses different actuators to manipulate the environment. An example of how this can be achieved is shown in listing 4.4 where the *on* behavior is defined to use a different implementation for that behavior. Reassigning behaviors in this way does not effect how the rest of the script operates.

```
1 on = "ie.tcd.cs.deskllighton"
```

Listing 4.4: Reassigning behaviors.

4.3.4 M Function

The primary function of the scripting language is to identify the set of contextual stimuli that influence an entity and to map them onto behaviors that allow entities to modify the local environment. In the stigmergic model, defined in chapter 3, the M function provides the means of mapping an entity's local environment onto B the behavioral set. To use the M function in the scripting language it is first necessary to identify the parts of the environment that act as stimuli to the entity. This is achieved by defining a set of predicates specifying the context information that is of interest to the entity. These are true when matched by information in the entity's current contextual view and can be used to determine the entity's behavior. The code shown in listing 4.5 is one such predicate.

In this sample code the context called *bobperson* is declared. The keyword *person* defines the predicate *bobperson* as identifying a part of the environment as a person with the name of *Bob*. The *location* keyword indicates a position or area that is of interest to the predicate, in

```
1 context bobperson
2 bobperson.person = "Bob"
3 bobperson.location = "O'Reilly House, F32"
4 bobperson.activity = any
5 bobperson.time = "lunch time"
6 bobperson.job = "teacher"
7 bobperson.music = "rock"
```

Listing 4.5: Declaring context predicate.

this case a symbolic location called *"O'Reilly House, F32"*. The *activity* keyword defines what is happening in that part of the environment. In this example the predicate is interested in Bob when doing any activity. The *time* keyword indicates a period, or point in time, which in this case is the symbolic time of *"lunch time"*. The last two lines from the sample code describe the job Bob does and specifies the type of music he likes to listen to.

As the above example indicates the scripting language uses a vocabulary based on Dey's [42] concept of primary and secondary context information to describe the parts of the environment that will act as the stimuli for regulating entity behavior. Primary context information being identity, location, activity and time, while secondary context information describes any other information that helps to define the situation. The scripting language uses this approach to define context though it extends it with additional scripting constructs to provide a more expressive means of describing the parts of the entity's local environment that will act as the stimulus for the entity.

```
1 somePredicate.place = "O'Reilly House, F32" // place
2 somePredicate.person = "Bob" //person
3 somePredicate.object = "Light" //object"
```

Listing 4.6: Assigning identity context.

Listing 4.6 shows how a predicate can identify the parts of the environment that are of interest to the entity. In this case it may be a certain place, as in the first example shown in listing 4.6, or a particular person as indicated in the second line of listing 4.6, and lastly, it may be an object in the environment that is of interest to the entity. The keywords *place*,

person, and *object* allow the scripting language to specify these preferences in a predicate.

```
somepredicate.location = "53°20'1"N, 6°15'1"W" //absolute
somepredicate.location = "10,15:ref=53°20'1"N, 6°15'1"W" //relative
somepredicate.location = "O'Reilly House, F32" //symbolic
```

Listing 4.7: Assigning location context.

A predicate can also express interest in a particular location or area through the use of the *location* keyword as shown in listing 4.7. The location can be specified in terms of a GPS coordinate as in the first example shown in listing 4.7, or as a relative coordinate as indicated to in the second example, and also in terms of a symbolic location as in the last example in listing 4.7. The relative coordinate is based on a reference point that acts as the origin and an x/y coordinate. The x/y coordinates are given in meters and indicate the offset from the reference point. The reference point can be a GPS coordinate, another relative coordinate, or a symbolic location. A symbolic location provides an abstract view of a location that typically maps to a GPS coordinate or a relative location, though it can also map to a specific area as in listing 4.7. It is possible to define either a polygon or circular area as can be seen in listing 4.8.

```
somepredicate.location = "0,0,1,1,0,1:ref=53°20'1"N, 6°15'1"W"//ploygon
somepredicate.location = "20:ref=53°20'1"N, 6°15'1"W" //circle
```

Listing 4.8: Assigning an area for a location context.

A polygon is defined by pairs of coordinates and a reference point that acts as the origin for the coordinates. A circular area is defined by a radius and a reference point that forms the center of the area. Expressing an area in these forms can be difficult and in practise it is normal that a symbolic location is used instead to map these locations. It should be noted that while symbolic context information can be used, the vocabulary needs to be agreed upon beforehand to the extent that symbolic information is matched exactly by the framework. How this is achieved is described in more detail in chapter 5.

Listing 4.9 shows how a predicate can identify activities in the environment that are of

interest to the entity. In this case an activity context is defined in terms of a string literal as listing 4.9 indicates. The significance of this value to the behavior of the entity is left to the developer to determine.

```
1 somepredicate.activity = "jumping"
```

Listing 4.9: Assigning activity context.

It is also possible to define a certain period, or point in time that an entity is interested in. This allows the scripting language to ignore parts of the environment until the appropriate time. A predicate can express this notion of time through the use of the *time* keyword as shown in listing 4.10. Time can be specified in terms of an absolute value as shown in the first example of listing 4.10, a relative time as indicated in the second example, or as symbolic value as in the third example, and also in terms of an interval or period of time as shown in the last example of listing 4.10. Relative time is based on a reference point and an offset that defines the point in time. The reference point can be an absolute value, as shown in listing 4.10, or another relative time, or symbolic time. A symbolic time provides an abstract view of time that typically maps to an absolute value or a relative value, though it can also map to a period or interval of time. An interval of time is defined by a reference point and value that denotes the duration of the period. The reference point, in a similar fashion to a relative value, can either be an absolute, relative, or symbolic time.

```
1 somepredicate.time = "Thu Mar 18 21:58:36 GMT 2004" //absolute
2 somepredicate.time = "10000:ref=31 August 2005 15:26:35" //relative
3 somepredicate.time = "lunch time" //symbolic
4 somepredicate.time = "ref=31 August 2005 15:32:58,20000" //interval
```

Listing 4.10: Assigning time context.

In the scripting language secondary context information is declared by specifying any key/value pairing. In the coding sample shown in 4.5 the *bobperson* predicate specifies two such pieces of context information. The first describes the job Bob does and the second specifies the type of music he likes to listen to. While defining secondary context in this way

provides a flexible approach for describing such context the key/value pairings can be open to interpretation by entities unlike the primary context information which is well defined. To be able to use such context it is necessary to agree beforehand the meaning to the extent that it can be matched by the framework.

```
1 somepredicate.person = any //any person
2 somepredicate.object= any //any object
3 somepredicate.place = any //any place
4 somepredicate.location = any //any location
5 somepredicate.activity = any //any activity
6 somepredicate.time = any //any time
7 somepredicate.job = any //secondary context - any job
8 somepredicate.music = any //secondary context - any music
```

Listing 4.11: The any keyword.

In the declaration of predicates the scripting language also defines the *any* operator. The *any* operator allows the predicate to assign any value to a context and for that to hold true when matching it with the entity's current contextual view. For instance, in the first example in the listing 4.11 the *any* operator is used to show the entity is interested in any part of the environment that is a person, or in the second example the entity is interested in any object that is in its environment, and in the third example shown in listing 4.11 the entity is interested in any part of the environment that can be thought of as a place. The *any* operator can also be used to show the entity is interested in a particular part of the environment that is at any location, doing any activity, or at any time, and to indicate that the entity is interested in a part of the environment with a secondary context of any value. The use of the *any* operator allows the scripting language to generalise on aspects of the environment rather than specifically stating what interests it. This allows the scripting language to reason about parts of the environment it might not know beforehand.

Once the required context predicates have been declared it is necessary to map the entity's contextual view on the behavior set by identifying the stimuli in the local environment that effect the entity's behavior and determining how the entity should modify its behavior in response. How this is achieved in the scripting language can be seen in listing 4.13. In this

```
1 context darkroom
2 darkroom.place = "O'Reilly Institute, F32"
3 darkroom.activity = any
4 darkroom.lighting = "dark"
```

Listing 4.12: Declaring context information for a dark room.

case, the mapping is accomplished when the context predicates *bobperson* (listing 4.12) and *darkroom* (listing 4.5) are found to be matched in the entity's current contextual view. This would indicate that Bob is in a place called *"O'Reilly Institute, F32"* with little light. On obtaining a match for this predicate the behavior can then be triggered for the entity, which in this case is the *on* behavior for a light. The general structure of the mapping statement allows the developer to specify one or more context predicates that all must hold in the entity's current contextual view for the mapping to be successful. All completed mappings map to one or more behaviors in the entity's behavioral set B. These are specified in the mapping statement by the developer.

```
1 map[bobperson, darkroom] onto{
2     on()
3 }
```

Listing 4.13: Example of mapping statement.

The declaration of context predicates and the definition of mappings for an entity can also be inherited from base scripts. For instance, a script could inherit context predicates *bobperson* (listing 4.12), and *darkroom* (listing 4.5) and use them along with other context predicates it has declared to define mappings. It is also possible, if a particular inherited predicate does not suit, to assign different values to a predicate. An example of how this can be achieved is shown in listing 4.14 where the location context on the *bobperson* is changed from *"O'Reilly Institute, F32"* to *"O'Reilly Institute, F35"*. It should be noted that reassigning values of inherited context predicates also effects how the mappings defined in the base scripts are performed. While this is a desirable attribute which allows a script to modify how mappings

are triggered in the base scripts care needs to taken to avoid unwanted behaviors.

```
1 bobperson.location = "O'Reilly House, F35"
```

Listing 4.14: Changing the value of a context predicate.

Mappings are inherited from base scripts in the same way as behaviors and context predicates and can be used by the extended script to dictate how the entity is to behave along with the other mappings defined in the script. Tailoring how the inherited mappings operate is achieved either by modifying the values of the predicates or by overwriting the mappings to change the behaviors that are mapped.

```
1 map[bobperson, darkroom] onto{
2   halfon()
3 }
```

Listing 4.15: Overwriting a mapping.

The sample code shown in listing 4.15 provides an example of how to overwrite a mapping to change the behaviors that are triggered. In this case, the mapping shown in listing 4.13 is overwritten to change the behavior it triggers from *on* to *halfon*. This is achieved by using the same context predicates in the mapping statement as in the inherited mapping. It is important to note that the context predicates have to match exactly otherwise the mapping is considered separate from that of the inherited mapping and as such will not be overwritten. The semantics for overwriting an inherited mapping with a mapping that provides a partial match has not been defined at this time.

4.4 Mapping

The previous sections have outlined the basic structure of the scripting language and have demonstrated how to trigger a behavior for an entity on encountering specific stimuli described by fragments of context information. The recognition of one particular instance in time is often not sufficient to capture the broader sense of what has occurred and it was necessary

to find a more expressive means of performing the mappings that can also take into account what has been observed beforehand. Influenced by the work of Allen [5] and that of Pinhanez et al's interval scripts [124] the section looks at another method that models the relationships between intervals of time to capture these observations and define entity behavior.

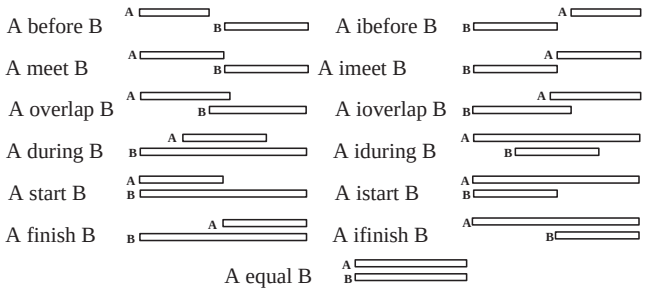


Figure 4.3: Interval relationships

4.4.1 Allen's Temporal Intervals

An interval of time is a length of time marked off by two distinct points in time representing the start and end of the interval. In [4, 5, 6], Allen introduced a model that made it possible to describe the relationship between two intervals of time. He showed that there are 13 such possible relationships, as summarised in figure 4.3.

Given any two intervals of time it is possible to use one of the relationships illustrated in figure 4.3 to describe how they are related. For instance, in taking a story such as the one below:

> John was not in the room when I touched the switch to turn on the light.

it is possible to use Allen's interval temporal logic to describe the above story as:

```
S overlap or meet L
S is before, meet, is imeet,
or ibefore R
```

where S is the time of touching the switch, L is the time the light was on, and R is the time that John was in the room.

The importance of Allen's work stems from its ability to provide a means of describing the relationships between intervals without having to explicitly mention the interval duration or specifying the relationships between the interval's extremities. These characteristics are of value when it comes to capturing the broader sense of what is happening in an environment. It can be used by the scripting language to describe the temporal relationships between observations so when the described relationship is satisfied the mapping can be triggered to modify the behavior of the entity. The method is especially useful when you also consider the the imprecise nature of the environments in which the entities are anticipated to operate.

4.4.2 Scripting Temporal Intervals

Based on Allen's interval temporal logic the script uses the primitive relationships defined by Allen to describe temporal relationships between intervals of time. Entity behavior is then triggered on observing the intervals in the correct temporal sequence. The context predicates described in section 4.3.4 are used to define the duration of the interval. The start of an interval is determined when the context predicate becomes true, and the end is denoted on it becoming invalid. The interval is deemed active between these two distinct points in time. The script specifies the relationships between intervals by defining a sequence of context predicates. Once the intervals have occurred, as indicated by the script, the mapping can occur and the behavior can be triggered.

```
1 //contextA start contextB
2 map[contextA, contextB][contextB]onto{
3     ...
4 }
```

Listing 4.16: Mapping using temporal intervals.

For the purpose of illustration an example is used to explain in more detail the use of Allen's interval temporal logic in the script. The sample code shown in listing 4.16 demonstrates

the use of intervals in a mapping statement. It uses the context predicates, *contextA* and *contextB,* to describe two different intervals of time. The relationship between the intervals can be defined as *contextA start contextB,* as per Allen's interval temporal logic. The square brackets demarcate the start and end of the intervals, and defines the relationship between them. Figure 4.4 shows graphically the relationship between the intervals.

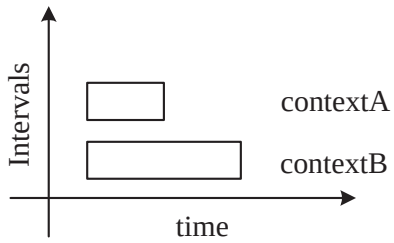


Figure 4.4: Interval - contextA start contextB.

In determining whether a mapping has been triggered the framework investigates each subsequent contextual view to determine if the intervals are active. An interval is deemed active when the context predicate is found to hold true in the entity's current contextual view and each subsequent contextual view after that. The interval becomes inactive when the predicate can no longer be found to be true. When the intervals are found to be active in the correct temporal sequence, as described in the mapping, it is then that the mapping can be triggered and the behavior of the entity changed to reflect the state of the local environment. In the example above, the relationship is satisfied when both *contextA* and *contextB* become active in the same instance with *contextB* remaining active for a period after *contextA* becomes inactive at which point the mapping can be triggered. When a mapping can actually be trigger is dependent on the interval and when it can be fully detected. This may only happen after all the intervals have ended, as is the case for the *finish* interval.

It is also possible to use the remaining 12 relationships defined by Allen to create different mappings. For instance, in the example shown in listing 4.17 *contextA is before contextB*. In this case, the temporal relationship is satisfied when the interval defined by the predicate

```
1 //contextA before contextB
2 map[contextA][][contextB]onto{
3     ...
4 }
```

Listing 4.17: Mapping statement contextA before contextB.

contextA is active in advance of *contextB* becoming active, in that, there exists another period of time that spans the time between the intervals as the empty square brackets indicate.

```
1 //contextA meet contextB
2 map[contextA][contextB]onto{
3     ...
4 }
```

Listing 4.18: Mapping statement contextA meet contextB.

In the sample code shown in listing 4.18 the mapping statement triggers when the temporal relationship *contextA meet contextB* is observed. This occurs when the interval defined by the context predicate *contextA* ends at the same point in time that the interval *contextB* becomes active. The mapping can be triggered at this point as the relationship has been satisfied.

```
1 //contextA overlaps contextB
2 map[contextA][contextA,contextB][contextB]onto{
3     ...
4 }
```

Listing 4.19: Mapping statement contextA overlaps contextB.

The listing 4.19 shows an example of a mapping statement using the temporal relationship *contextA overlaps contextB.* In this case, the mapping will only be triggered when the end point of the interval specified by the predicate *contextA* overlaps the start point of the interval defined by *contextB.* At this point the mapping can be triggered and the behaviors for the entity can be invoked.

The sample code shown in listing 4.20 illustrates the use of the temporal relationship *con-*

```
1 //contextA during contextB
2 map[contextB][contextA, contextB][contextB]onto{
3     ...
4 }
```

Listing 4.20: Mapping statement contextA during contextB.

textA during contextB within a mapping statement. To satisfy this relationship the mapping must observe that the interval defined by the predicate *contextA* occurs after the start of the interval defined by *contextB* but before the interval is inactive.

```
1 //contextA finish contextB
2 map[contextB][contextA,contextB]onto{
3     ...
4 }
```

Listing 4.21: Mapping statement contextA finish contextB.

In the sample code shown in listing 4.21 the mapping statement triggers when the temporal relationship *contextA finish contextB* is observed. This occurs when the interval defined by the context predicate *contextB* becomes active before *contextA* but ends at the same point in time as *contextA* becomes inactive. It is at this point that the framework can trigger the mapping and invoke the behaviors associated with it.

```
1 //contextA equals contextB
2 map[contextA,contextB]onto{
3     ...
4 }
```

Listing 4.22: Mapping statement contextA equals contextB.

The listing 4.22 shows an example of a mapping statement using the temporal relationship *contextA equals contextB.* For this mapping to be triggered the intervals defined by predicates *contextA* and *contextB* must both be active at the same time. In other words, the intervals most both start and end at the same point in time. The mapping for this temporal relationship

can only be triggered when the framework observes both intervals becoming inactive at the same time.

```
1 //contextA ibefore contextB
2 map[contextB][][contextA]onto{...}
3 //contextA imeet contextB
4 map[contextB][contextA]onto{...}
5 //contextA ioverlaps contextB
6 map[contextB][contextB,contextA][contextA]onto{...}
7 //contextA iduring contextB
8 map[contextA][contextA,contextB][contextA]onto{...}
9 //contextA istart contextB
10 map[contextA,contextB][contextB]onto{...}
11 //contextA finish contextB
12 map[contextA][contextA,contextB]onto{...}
```

Listing 4.23: Other mappings using intervals.

The remaining relationships defined by Allen are the inverse of those used in the mapping statements shown in listings 4.16, 4.17, 4.18, 4.19, 4.20, 4.21, and 4.22, and as such operate in a similar though opposite way. These mappings are shown in listing 4.23.

```
1 //contextB start contextA, contextA finish contextB,
2 //contextA during contextC
3 map[contextB, contextC][contextA,contextB,contextC][contextC]onto{
4   ...
5 }
```

Listing 4.24: A more complex mapping statement.

More complex mappings can also be achieved by using multiple intervals as in the example shown in listing 4.24 and illustrated in figure 4.5. The relationships between these intervals can be defined as follows: *contextB start contextA*, *contextA finish contextB*, *contextA during contextC*.

The use of Allen's interval temporal logic provides the scripting language with a more expressive means of performing mappings that takes into account what has occurred beforehand and not just what has occurred at a single point in time. While it does increase the complexity of the script, it is felt that the increased expressiveness gained by using Allen's temporal

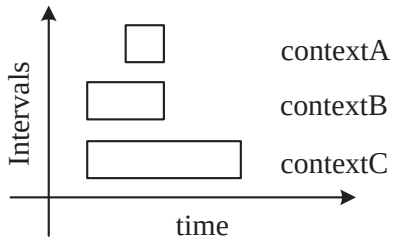


Figure 4.5: Interval - contextB start contextA, contextA finish contextB, contextA during contextC.

logic outweighs the additional difficulty in scripting the behaviors of entities. It should also be noted that the mapping statements described in this section use the same semantics for inheritance as those described in section 4.3.4.

4.5 Tailoring Entity Behavior

The previous sections introduced the concept of using temporal intervals to trigger the different behaviors of an entity. This provides entities with the ability to react to different contextual stimuli within the entity's local environment. In tailoring this process the scripting language provides a method for customising how the behavior is invoked. This allows the scripting language to focus an entity's behavior on a particular part of the environment within the entity's current context contextual view. It also allows entities to more accurately adjust how the behavior is triggered to better suit different situations that might be found in the environment.

4.5.1 Passing Context Information

The parameters for the behaviors are constructed from the context information which has been defined beforehand within predicates declared in the script, or from context information that is held in the entity's current contextual view. The sample code in listing 4.25 demonstrates how this can be achieved in the script language. The context information *peter.music* is passed

to the *play* behavior on being invoked by the framework. For context information to be passed the behavior must first be implemented to take a parameter. The context information passed must also match what is expected by the behavior otherwise the invocation will fail. It is also possible to pass the behavior more than one parameter though it needs to be specified before in the implementation of the behavior. Further details of how the framework passes context to behaviors are described later in chapter 5.

```
1 context peter
2 peter.location = "O'Reilly Institute, F32"
3 peter.music="folk"
4 ...
5 play(peter.music)
```

Listing 4.25: Passing context information to a behavior.

4.5.2 Using Embedded Functions

To access information held in the entity's contextual view the scripting language uses embedded functions that allow the data in the contextual view to be analysed and context information deduced without having to directly handle the data. The implementations of the embedded functions are not provided by the scripting language but in Java following a particular API defined by the framework. The specification of the API is covered in section 5.2.4.3. To use a particular embedded function it must first be declared in the script or within one of the base scripts of the script. An example of how a embedded function is declared can be seen in listing 4.26. In this example the embedded function *majority* is declared with a reference to the Java classes that implement the function.

```
1 //Declaring embedded function majority.
2 efunction majority = "ie.tcd.cs.dsg.cocoa.Majority"
```

Listing 4.26: Declaring embedded function.

The example shown in listing 4.27 illustrates how such an embedded function can be used

in a script. The *majority* function, in this case, allows a developer to determine the value of a piece of context information that is most prevalent in the entity's current contextual view. In the example shown in listing 4.27 the context *someperson* acts as a filter for the *majority* function excluding any entities in the current contextual view that are not in the location "*O'Reilly Institute, F32*" or do not have any music preference. From the remaining context the *majority* function is used to determine the majority value of the secondary context *music* which is then passed as a parameter to the *play* behavior.

```
1 //Declaring context predicate someperson.
2 context someperson
3 someperson.location = "O'Reilly Institute, F32"
4 someperson.music=any
5 ...
6 //Using majority function to determine parameter for behavior
7 play(majority(someperson))
```

Listing 4.27: Using embedded function.

The context used by an embedded function, and which eventually gets returned, is discovered through the flow in which the function is being called. This is determined when the script is initially loaded and interpreted. In the example taken, the majority function is called within the play behavior which expects a parameter of secondary context music. The majority function uses the discovered information to focus on the context to provided, in this case, the play behavior with the majority value of the secondary context music that is currently in the entity's contextual view.

The scripting language currently supports a number of embedded functions. These include the *majority* function mentioned above, and also the *minimum*, *random*, *minority*, and *average* embedded functions. These functions are declared in the *entity* script (see figure 4.1) so that they may be accessed by all scripts. Developers can still extend the functionality of the scripting language by either implementing their own embedded functions or changing the behavior of an embedded function by reassigning a different implementation of that function as can be seen in listing 4.28.

The *maximum* and *minimum* functions allow the developer to retrieve the maximum or

```
1 //Assign new implmentation to majority function.
2 majority = "ie.tcd.cs.dsg.cocoa.MajorityNewImpl"
```

Listing 4.28: Reassign embedded function.

minimum values for context information in the entity's current contextual view. The *random* function randomly selects a piece of the context information in the entity's current contextual view. Using *majority* function allows the developer to determine the value of a piece context information that is most prevalent in the entity's current contextual view. The *minority* function provides opposite functionality to that of the majority function. The *average* function determines the average value for a particular context.

Currently the embedded functions are restricted to deriving the parameters to be passed to behaviors. The embedded functions cannot be used to assign values to context predicates as to do so would create a situation where the intervals are constantly being redefined. This would lead to inconsistencies in determining whether a temporal relationship defined in a mapping, or in this case redefined, has been satisfied or not.

```
1 //Determining parameter for someBehavior using
2 //functions EFunction1 and Efunction2.
3 someBehavior(EFunction1(EFunction2(somePredicate)))
```

Listing 4.29: Calling one embedded function within another.

It should also be noted that an embedded function can be called within another as the sample code shown in listing 4.29 illustrates. However, for the embedded functions - *minimum*, *random*, *majority, minority*, and *average* - currently implemented for the scripting language it does not make sense to use this functionality as they typically only return a single context value. The benefit comes from embedded functions that return more than one value. In these cases two or more functions can be used to achieve the desired result.

4.5.3 Using the *This* Keyword

The behaviors described in a script characterise how a particular type of entity behaves in the environment and can be reused by all entities of that type to define their behavior. The approach allows the framework to generalise the behaviors of entities and ensures there is greater reuseability. However, in certain circumstances it is necessary to reference the entity that is the subject of the script. To do so explicitly would impair the ability of the framework to reuse code. There is a need to provide mechanism a for referring to this entity in the script. The *this* keyword provides such functionality. It provides a reference to the contextual state of the entity initialised with the script.

```
1 //Declaring context predicate LightOff using the this keyword.
2 context LightOff
3 LightOff.object = this.object
4 LightOff.activity = "Off"
```

Listing 4.30: Using the this keyword to define context predicates.

In listing 4.30 the *this* keyword is being used to define the context predicate *LightOff*. In this case the predicate uses the *this* keyword to identify the entity who is the subject of the script. The same approach can also be used to pass context information to behaviors as shown in listing 4.31.

```
1 //Mapping statement - contextA start contextB
2 map[contextA, contextB][contextB]onto{
3   lighton(this.object) //Entity behavior call lighton
4 }
```

Listing 4.31: Using the this keyword to pass context information.

4.5.4 Redefining P

While the proximity function, P, is usually set at the beginning of the script it can, if the circumstances required it, be modified during the lifetime of the entity through additional calls

to the proximity function. Modification of the proximity function are sometimes required by entities when moving from one environment to another. For example, when a PDA moves from a busy street to a office it may redefine P to take into account its new environment. Listing 4.32 provides one such example of the proximity function being redefined. Typically, redefining the proximity function is done within the mapping statement as the example above illustrates.

```
1 //Mapping statement - contextA finish contextB
2 map[contextB][contextA,contextB]onto{
3   displayPicture() //Entity behavior called displayPicture.
4   proximity(8) //8 meter radius around entity.
5 }
```

Listing 4.32: Redefining P.

4.6 Summary

The chapter has described a high-level scripting language for scripting entity behavior in pervasive computing environments. Based on the stigmergic model developed in chapter 3 the scripting language provides a programming model that combines expressiveness and simplicity with the ability to abstract the complexities of dealing with the underlying technologies. By focusing on defining entity behavior the scripting language allows developers to concentrate their efforts on characterising the behavior of a pervasive computing environment rather than system development while also aiding the incremental construction and improvement of solutions over the lifetime of the environment.

In using a programming model based on a high-level scripting language the goal has also been to develop an approach that provides support for requirement R1 specified in section 3.1.2 - supporting the physical integration of components into the environment. From the sample code presented in the chapter and descriptions provided it can be argued that the scripting language provides the high-level abstractions sought by this requirement and which allow components to sense and interact with the physical environment without the difficulties

of dealing with low-level devices such as sensors and actuators.

The next chapter takes the stigmergic model developed in the previous chapter and the scripting language proposed by this chapter and provides details of their implementation.

Chapter 5

Cocoa Framework

Previous chapters have introduced an alternative approach to constructing pervasive computing environments and have established a set of requirements for supporting their development. Chapter 3 proposed a model capable of meeting these requirements. Based on the principles of stigmergy it describes a method of building the type of pervasive computing environments envisioned in section 3.1.1. The previous chapter builds on this work by introducing a programming abstraction encapsulated in a high-level scripting language. It is based on the stigmergic model defined in chapter 3 and provides a means for defining entity behavior in a pervasive computing environment.

This chapter presents a prototypical implementation of the stigmergic model and of the scripting language used to define entity behavior. The current prototype is called Cocoa (COordinated COntext Awareness) [8] and has been implemented on Linux[1] using Java[2]. The chapter begins with an outline of the architecture of Cocoa and discusses the functionality provided by the main components of the framework as well as the means by which these components are used to develop individual entities. The chapter also describes the algorithms used and the possible system configurations that can be applied along with a demonstration that illustrates how individual entities are developed.

[1] Using version 9 of Redhat's linux distribution.
[2] Using Sun Microsystem's implementation, version J2SE 1.4.2.

5.1 Architectural Overview

In developing the Cocoa framework it has been necessary to incorporate technologies that both support and complement the use of stigmergy. The framework has been designed as a distributed architecture organised in a peer-to-peer fashion. Each node in the architecture represents an entity in the pervasive computing environment. A modular design is used to aid both the extensibility and flexibility of the framework. This allows different components to be loaded at runtime depending on the entity. Each entity runs in its own computational space. Entities may be located on the same device but they do not necessarily have to be.

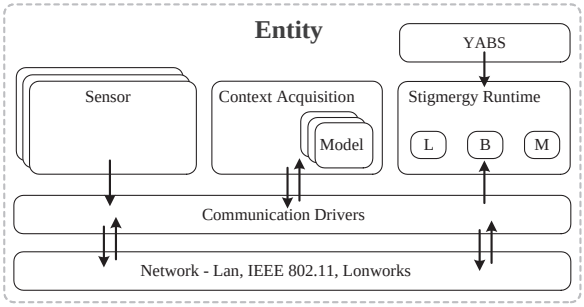


Figure 5.1: Cocoa Architecture

The architecture of Cocoa, as shown in figure 5.1, primarily consists of five key components. These reflect the main functional parts of the framework. Sensor components provide the interface to the physical sensors associated with the entity. Each sensor component represents a sensor in the environment. Different sensor configurations can be used depending on those loaded into the framework at runtime. The sensor components provide the framework with the ability to sense the physical environment. The data retrieved from the sensors is provided to the context acquisition component. It uses the sensor data to capture context information concerning the local environment. An open interface is provided by the component to allow different techniques or *models* to be plugged into the framework to interpret the sensor data. The choice of model is determined at runtime to allow the framework match the model to the entity's requirements.

As highlighted in section 3.4, the acquisition of context information uses a collaborative

approach, whereby each entity acquires the contextual state of the physical environment around them and publishes it in their local environment where it can be used by other entities. The process helps to distribute the cost of capturing context information as the entities can share the information they have derived from sensors. The stigmergy runtime collects the context information produce by the context aquisition component of the entity and of the surrounding entities to determine the state of the local environment. The collated information is then used by the stigmergy runtime to decide the future behavior of the entity. The component applies the three primitives - the proximity function L, the behavioral set B, and the M function - described in section 3.5 to define the mechanisms for determining the state of the local environment and behavior of each entity.

The communication drivers support a decoupled communication model that distributes context and sensor data events. The current implementation allows different communication drivers to be plugged into the framework to suit both the middleware requirements (for example quality of service, or real-time requirements) or network configuration (for example wireless ad-hoc networks, or wired Ethernet networks). This makes it easier to modify the framework to suit a variety of environments. Binding the framework together is the scripting language defined in chapter 4. It produces the intermediate objects used by the stigmergy runtime to manage the entity's behavior. A more detailed description of the main components is provided in the next section.

5.2 Main Components

This section describes the design choices and implementation details of the main components - communication drivers, sensors, context acquisition, stigmergy runtime, scripting language - of the framework and their relationship to the stigmergic model defined in chapter 3.

5.2.1 Communication Drivers

In the current implementation it is possible to plug in different communication drivers to suit both the middleware requirements and network configuration. This provides a degree of

flexibility that allows the framework to modify the configuration of the entity to suit different environmental conditions. The communication drivers can be based on publish subscribe mechanisms, tuple spaces, or other communication paradigms that provide a decoupled communication model. This type of communication model is required to maintain the indirectness between entities and ensure requirements R3: spontaneous interoperability, and R6: robust behavior (section 3.1.2) can still be satisfied. The current implementation provides two drivers, one based on STEAM [99, 100] and another on Siena [29].

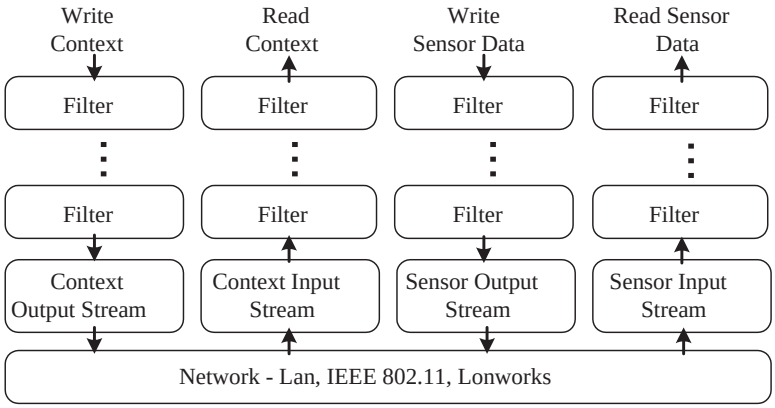


Figure 5.2: Structure of Communcation Driver.

5.2.1.1 Context and Sensor Data Channels

The communication drivers are used to share environmental sensor data and context information between entities in the local environment. To achieve this the driver provides two data channels, one for sending and receiving context information, and another for sensor data. Figure 5.2 provides an overview of the driver. In each of these channels there are input and output streams for the data. It is possible to add filters to either of these streams. This allows the framework to pre-process the data before it is written to or read from the stream, allowing the framework to add extra functionaility. The order in which these filters are inserted, or what communication driver is used is decided at runtime when the components are initially loaded into the Cocoa framework. Section 5.3 describes in more detail the initialisation process and

the possible configurations that can be achieved.

```
1 //Error code omitted to simplify presentation.
2 //Get context communication driver from framework.
3 ContextComms conComms;
4 conComms=(ContextComms)Cocoa.components.retrieve(Cocoa.CONTEXT_COMMS);
5
6 //create a new piece of context information.
7 Context context = new Context(...);
8
9 //writing context information to channel.
10 contextComms.write(context);
```

Listing 5.1: Writing context information to channel.

Listing 5.1 provides an example of how the higher level components - sensor, context acquisition, and stigmergy runtime - are able to use the communication driver. In this code excerpt a cocoa component retrieves the context information channel and writes a piece of context information to the channel. A similar process is use to write sensor data.

```
1 //Error code omitted to simplify presentation.
2 //Get sensor communication driver from framework.
3 SensorComms senComms;
4 senComms=(SensorComms)Cocoa.components.retrieve(Cocoa.SENSOR_COMMS);
5
6 //Read sensor data from channel.
7 SensorData data = senComms.readSensorData();
```

Listing 5.2: Reading sensor data from channel.

In listing 5.2, a component obtains the sensor data channel from the framework and uses it to retrieve sensor data from the channel. Again, a similar method is used by components to obtain context information.

5.2.1.2 Concrete Driver Implementations

The Cocoa framework defines an abstract implementation of the communication drivers. Specific concrete implementations of the communication drivers need to be provided. The current

prototype provides two implementations, one, which is based on the STEAM event service [99, 100], and another that is based on Siena [29].

STEAM is an event-based middleware service that has been designed with pervasive computing in mind. More specifically, it is intended for use within mobile environments using wireless ad-hoc networks. STEAM exploits a number of novel techniques that allow it to operate successfully within these types of environments. In particular, it uses geographical information to limit the propagation of events through the environment to improve scalability and achieve timely delivery of events. There are also no centralised components within STEAM and subscription to events are made dynamically to nearby producers as entities move through the environment. Events can also be filtered based on the proximity of one entity to another, which bounds the range within which event notifications are delivered. The STEAM event service was chosen as it particularly suits the mechanisms that Cocoa promotes and the type of environment in which it is anticipated that it will be deployed. A detailed description of STEAM can be found in [99, 100].

Siena (Scalable Internet Event Notification Architecture) [29] was developed at Politecnico di Milano and has been designed to provide a scalable, general-purpose distributed event-notification service for internet scale applications. The system aims to provide an expressive approach to defining events and filters without sacrificing scalability. Event clients interact through a distributed hierarchy of servers whose purpose is to route events between clients. Siena is a very different event service to STEAM which is specifically designed to operate in mobile ad-hoc environments. However, the current implementation of STEAM has difficulty in operating properly within enclosed environments due to its reliance on GPS coordinates for positioning. Siena can operate in these environments so provides the necessary communication driver for indoor use and simulations.

The concrete implementations of the communication drivers are responsible for marshaling and unmarshaling the sensor data and context information events to the underlying communication system. The type of events used and the mechanisms employed in converting the data to and from the events is specific to each of the drivers, as are the configurations used in initialising the drivers. In the case of the STEAM communication driver three event types are

used, one for sensor data, a second for sensor metadata, and a third for context information. The data is marshaled into key-value pairings that correspond to the structure of the data being transmitted. Details of the sensor data and context information used are provided in later sections. In contrast, Siena only uses two event types, one for sensor data and metadata, and a second for context information. In this case the sensor data and context information objects are serialised into a byte stream using Java's object serialisation. The result is placed into an event, which can then be deserialised by entities receiving the events.

The configurations for either the STEAM or Siena event services are obtained from a configuration file read in at runtime by the framework, details of which are provided in section 5.3. For Siena this includes the nearest Siena server that the framework can use to publish and subscribe to sensor and context events. For STEAM it establishes the mode of operation for the event service - either mobile or fixed. In fixed mode the entity is expected to be static, while in mobile mode the STEAM consumer or producer is anticipated to move through the environment. It also sets the range for the proximity filter for the three event types. Proximity filters are used in STEAM to bound the range within which event notifications are delivered.

5.2.2 Sensors

In the stigmergic model, described in chapter 3, environmental sensors were shown to be an integral part of an entity's ability to sense its surrounding environment. The data obtained from the sensors allows entities to determine the contextual state of their local environment. The changes detected in this environment give entities the ability to coordinate their behavior with other entities in the pervasive computing environment. In this model entities are connected permanently to a collection of sensors that allow them to sense the basic parameters of their environment. As they move through the environment they can also use the data from sensors connected to other entities. The sharing of sensor data at this level allows entities to cooperate in obtaining a better understanding of their local environment. To achieve this the cocoa framework provides a sensor component for environmental sensors. It is an abstract implementation of a sensor providing the basic functionality required for an entity to use the sensor and to share the sensor data with other entities in the local environment.

Specific implementations of this component are required for each type of sensor used by the framework.

5.2.2.1 Sensor Component

Figure 5.3 illustrates the main parts of the sensor component. The *SensorComponent* is an abstract class providing the basic functionality required for a sensor to be loaded into the framework. It inherits from *AbstractSensor* which provides methods for accessing and retrieving the data from the sensors. To use a particular environmental sensor in the framework a concrete implementation of the *SensorComponent* is required for the sensor to work. The sensor can then be loaded into the framework at runtime and made available to the entity.

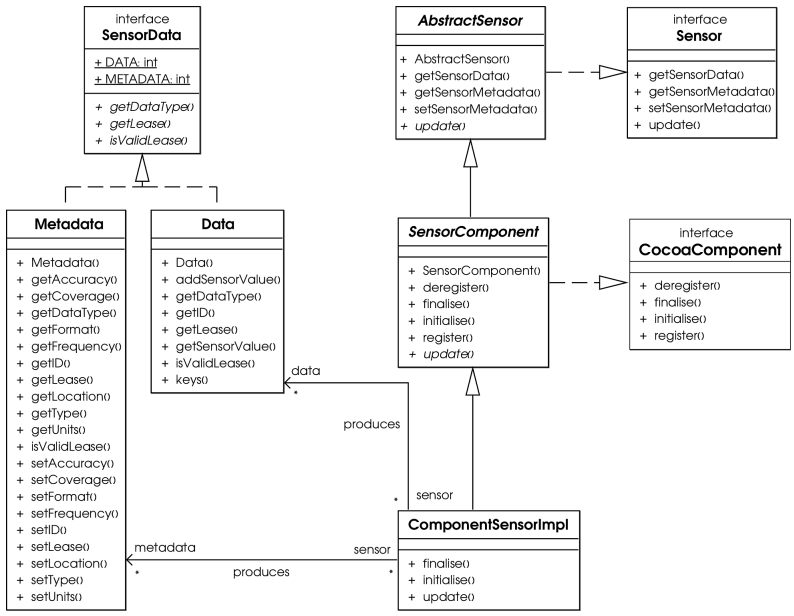


Figure 5.3: Class Diagram of Sensor Component.

The sensor component produces two types of sensor data, the first type - *Data* - represents an actual sensor reading obtained from the sensor, the second type - *Metadata* - describes the sensor, its location, the coverage of the sensor, the accuracy of the sensor, and type data produced by the sensor. The sensor metadata is periodically disseminated through the sensor channel of the communication driver to allow entities discover the sensor. The information

allows entities to select the right sensor to use and to interpret the data received from the sensor. The sensor data is also written to this channel when readings from the sensor have changed or after a period of time.

The sensor component uses a lease-based mechanism for both types of sensor data. This is used to overcome the possibility of failure or disconnection from the sensor and to prevent the use of stale sensor data in the context acquisition component. All sensor data written to the communication driver is marked with a lease that demarks the period of time for which the data is valid. It is also tagged with an identifier of the sensor producing the data. This is used to distinguish between different sources of data allowing the entity to manage and use the sensor data in a more efficient manner. The framework, typically, uses a 32-bit randomly generated number in combination with the metadata describing the sensor type to identify each of the sensors in the environment. It is excepted that this should provide an adequate method of avoiding duplicate sensor identifiers within the same local environment.

As the sensor component produces sensor data it guarantees to reissue it before the lease runs out or to provide a new reading if it is available before that happens. In this way the context acquisition component is able to determine when a sensor is no longer available to it. It can then remove any of the sensor data it retains for that sensor. This approach ensures the freshness of the sensor data and prevents stale data being used in determining the contextual state of the local environment.

5.2.2.2 Sensor Data

The readings from the physical sensor are placed by the sensor component into a *Data* object. The *Data* object is designed to take one or more values for each sensor reading. For example, a reading from a GPS device can be represented by two values, one for latitude, and another for longitude. The sensor values are typed, in that, they can be represented as a *long*, *float*, *double*, *int*, *boolean*, or *string.* In the case of a GPS coordinate two double values can be used by the GPS sensor component to represent each of the sensor readings.

The code excerpt shown in listing 5.3 illustrates how the sensor readings are handled within the framework, first, by creating the sensor values, in this case, the longitude and latitude

```
1 //Error code omitted to simplify presentation.
2 //Create longitude and latitude values. Dublin in this case.
3 DoubleSensorValue lon = new DoubleSensorValue(53.6345);
4 DoubleSensorValue lat = new DoubleSensorValue(6.2543);
5
6 //Create Data object for GPS data.
7 //lease is the period for which the data is valid.
8 //sensorID is unique identifier of the sensor.
9 Data data = new Data(lease,sensorID);
10
11 //Add sensor values to data object.
12 data.addSensorValue("lon",lon);
13 data.addSensorValue("lat",lat);
```

Listing 5.3: Creating a GPS sensor reading.

values of a GPS coordinate, then by creating the *Data* object and specifying the lease and the ID of the sensor producing the coordinate, and lastly, by adding the sensor values to the data object. A similar approach is used for all sensor data produced by sensor components.

By structuring the sensor data in this way it is possible to support a diverse range of data formats, allowing a large array of sensors to be able to provide data in a format that can be presented to the other components in the framework. However, the generic format of the sensor data necessitates the need for addition information - metadata - to describe the data being provided. This is required if other entities in the environment are to be able to dynamically use the data produced by the sensors.

5.2.2.3 Sensor Metadata

The framework uses metadata to describe the sensors in the environment and the data being produced by them. The aim is to provide components with enough information for them to dynamically select and use the data coming from the sensors. To achieve this the framework provides components with a metadata model that can be used to characterise sensors and the data they produce. Figure 5.4 provides an overview of this model. It currently supports eight classifications that can logically be broken into two categories; those which characterise the sensor, and those that identify the format of the data produced. While this model does

not provide an extensive classification of sensors compared to SensorML [37] it is a proof-of-concept that is sufficient for addressing the needs of the Cocoa framework.

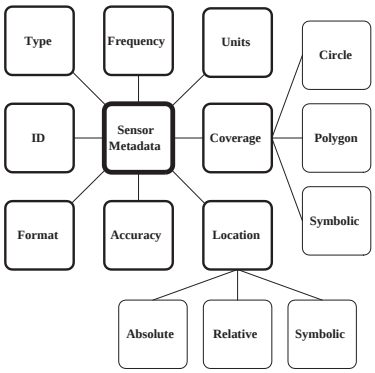


Figure 5.4: Sensor Metadata Model.

The *type* metadata identifies the variety of sensor data being produced by the sensor. For instance, a GPS device would generate *location* data, while a thermometer would produce *temperature* data. The data is used by the framework to choose particular types of sensor data to determine the context information of the local environment. The *frequency* metadata defines the sampling rate of the sensor. The information is useful in determining the speed at which the sensor data is being produced. The *units* metadata specifies the units of measurement used by the sensor. For example, a speedometer may have the data in miles per hour or kilometers per hour. The *accuracy* metadata indicates the degree to which the sensor component believes the sensor data to be a true reflection of the environment. It is expressed as a numerical value between 0 and 1. The *ID* metadata is the unique identifier for the sensor in the environment. The same identifier is used to tag all the sensor data produced by the sensor. In this way the framework is able to match the sensor data received with the metadata of the sensor.

The *format* metadata defines the order and type of values being used by the sensor component to disseminate sensor readings. It defines the format of the sensor data generated by the component. Other components in the framework use this information to dynamically interpret the data being produced by each of the sensor components. The code extract shown in

listing 5.4 provides an example of how the sensor *format* metadata is defined for a GPS sensor within the Cocoa framework. This sample code matches the GPS sensor reading generated in listing 5.3.

```
1 //Error code omitted to simplify presentation.
2 Metadata metadata = new Metadata(); //Create metadata object.
3
4 //Create format metadata for gps sensor data
5 String[] key = {"lon","lat"};
6 int[] type = {FormatSensorInformation.DOUBLE,
7                FormatSensorInformation.DOUBLE};
8 FormatSensorInformation fmat = new FormatSensorInformation(key,type);
9
10 metadata.setFormat(fmat); //Add to sensor metadata object.
```

Listing 5.4: Defining metadata for GPS sensor.

The *location* metadata specifies the physical location of the sensor. The location can be expressed in terms of an absolute, relative, or symbolic value. An absolute value is defined as a GPS latitude/longitude coordinate in the framework. A relative value defines the location of the sensor in relation to another object, or location. A symbolic value represents an abstract view of the sensor's location, for example, the sensor is located in *"O'Reilly Institute, F32"*, or attached to *"Jim"*.

The *coverage* metadata defines the extent to which the sensor can sense. For example, a thermometer located in the center of an office can be excepted to reliably sense the temperature of the whole office, implying the coverage of the thermometer to be the boundaries of the office. If you take a passive infrared (PIR) sensor the coverage can be expected to be a conical region in front of the sensor, in which, the sensor would detect the movement of a person. The metadata model allows the coverage of a sensor to be defined as either a circle, polygon, or symbolic area around the sensor.

The location and coverage metadata are probably the most important for an entity to discover a sensor in the environment. Together they can be used to determine the location of the sensor and from that ascertain whether the coverage is sufficient to allow the entity use the sensor data from the sensor.

5.2.3 Context Acquisition

In the stigmergic model, described in chapter 3, it is the context information from the local environment that is used as the common medium for the indirect communication between entities. To obtain context information entities use the data produced by the sensors and apply different sensor fusion techniques to derive the context information they required to determine the state of the local environment. The changes detected in context information provide entities with the stimulus that they require to coordinate their behavior with other entities in the pervasive computing environment. To achieve this the Cocoa framework provides a component for acquiring context information. Its purpose is to retrieve the sensor data, to apply the most appropriate techniques to derive the context information, and then to disseminate the context information to entities in the surrounding environment. The dissemination of the context information is done to support the cooperative model of sensing the environment.

5.2.3.1 Context Acquisition Component

The context acquisition component uses the sensor channel of the communication driver to retrieve data from sensors attached to the entity and from those sensors shared by other entities in the surround environment. The data gathered is used to determine context information for the local environment. An open interface is provided by the component so that different techniques or *models* can be plugged into the framework to interpret the sensor data. The models may use techniques based on simple IF-THEN rules, or other sensor fusion techniques such as Bayesian networks. Any of context information produced by the component is published to the context information channel of the communication driver.

Figure 5.5 provides an overview the main parts of the context acquisition component. The *ContextAcquistionComponent* is an abstract class providing the necessary functionality for the component to be loaded into the framework. It inherits from *AbstractContextAcquistion* which provides methods for retrieving the context information from the *Model* and for inserting the sensor data into the *Model*. The *ContextAcquistionComponent* also has responsibility for requesting the *Model* to be used from the *ModelManager*. The *ModelManager* administers all

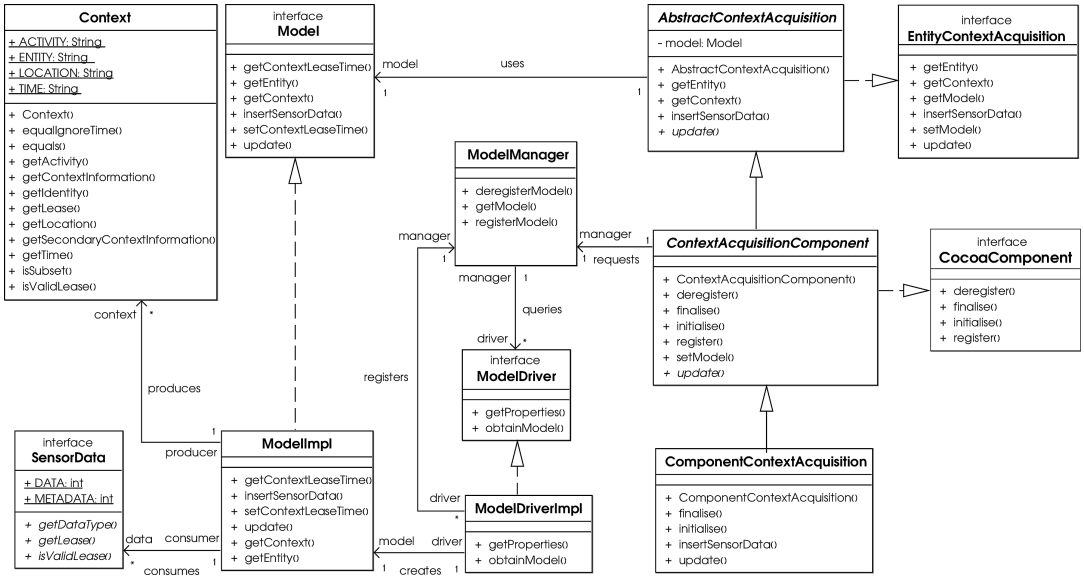


Figure 5.5: Class Diagram of Sensor Component.

the *ModelDrivers* for the framework. All *ModelDrivers* implementations - *ModelDriversImpl* - register with the *ModelManager* when loaded into the framework. The *ModelManager* queries each of the registered drivers to determine which provides support for current environmental configuration. If more than one is capable of providing support the *ModelManager* chooses the first one. The selected driver provides a specific implementation of a *Model* that can be used in the framework to derive context information from sensor data retrieved from the environmental sensors. The *ComponentContextAcquistion* provides a concrete implementation of *ContextAcquistionComponent*. It monitors the communication driver for sensor data and updates the model when the data is available. The component also disseminates any context information produced by the model to the entities in the surrounding environment.

It should be noted that the context acquisition component also uses a similar leasing mechanism to that of the sensor component. To prevent the use of stale information by the framework, in this case context information being used by the stigmergy runtime, all context information produced by the component is marked with a lease that demarks the period for which the information is valid. The context acquisition component guarantees to

reissue the information before the lease runs out or to provide new context information if it is available before that happens. In this way the framework is able to determine when a source of information is no longer available to it and so remove any of the information it retains from that source.

What is also important to note at this stage is that an entity only acquires context information for part of the environment. The wider contextual picture is gained from entities implicitly sharing their context information with other entities. This they achieve by listening to the context information channel of the communication driver which provides entities with all the context information generated in the surrounding environment. Using this approach simplifies the process of determining the contextual state of the local environment while still ensuring that entities remain decoupled.

5.2.3.2 Context Acquisition Models

The context acquisition models are pluggable components that can be inserted into the framework at runtime to interpret sensor data. Depending on the environment and the entity involved different models can be used by the framework to obtain context information for the local environment. While is it possible for these models to use a variety of techniques, the framework does not specify which to use, or indeed, provide any concrete implementations of them. This is left to the application developer to choose the most appropriate technique for the entity to use. By using this approach it allows the Cocoa framework to tailor the process of capturing context information for each entity. The modular aspect also ensures entities can adapt the process of acquiring context information to suit different environmental conditions. It is also possible for new techniques in capturing context information to be incorporated into the framework at a later stage.

As can be seen in figure 5.5 the model implementation - *ModelImpl* - consumes the data generated by the sensor components and produces context information that the entity can use and share with other entities in the surrounding environment. The technique used to derive the context information is incorporated within *ModelImpl*. It determines how the sensor data is used to derive the context information for the entity. However, as Mostéfaoui et al. [106]

have noted there are sources for context information other than those sensed by a sensor, for instance, derived context, where context information is computed on the fly. A typical example is that of time and of date. There is also context information that is explicitly provided either by the user or administrator. This can be though of as static context information that does not change significantly over a period of time. A good example are user preferences that have been explicitly communicated by the user. The Cocoa framework provides support for both sources for context information.

5.2.3.3 Dynamic Discovery and Use of Sensor Data

To dynamically discover and use the data from sensors in the way sought by the stigmergic model, presented in chapter 3, requires that a certain level of knowledge on each of the sensors in the environment can be obtained. The sensor metadata, outlined in the section 5.2.2.3, provides the means of achieving this. It allows the framework to describe a sensor and the data produced by it. This information is generated periodically by the sensor component and published to the sensor channel of the communication driver. The information acts to both advertise the presence of the sensor to entities and as a heart beat for the sensor.

In the current implementation of the context acquisition component all sensor metadata received from the communication driver is filtered and placed into a repository that can be queried at a later stage. The repository is periodically purged of metadata that is no longer valid. This is determined by the leasing mechanism outlined in section 5.2.2.1. As the sensor data is retrieved from the sensor channel, the context acquisition component is able to pre-process the data before it is passed to the model. It first filters the sensor data that does not have a corresponding metadata entry. This is determined by the identifier tagged to all sensor data and metadata. All remaining data is inserted into the model. At this stage the model can use the metadata - *location, coverage, type*, *frequency*, *accuracy*, and *units* - to select the sensor data to use in determining the context information of the local environment. For instance, the location and coverage metadata can be used to determine if the sensor is sufficiently close enough to use. The choice of sensor data to use is left to the application developer to decide.

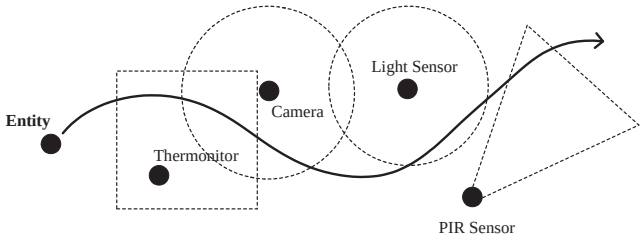


Figure 5.6: Dynamic selection and use of sensors.

The aim, as figure 5.6 shows, has been to provide a framework with the ability to dynamically discover and select what sensors to use as the entity moves through the physical environment. The use of sensor metadata provides a means by which this can be achieved. However, the cost of periodically announcing the sensor to the environment and the additional filtering of data by the context acquisition component may be too resource intensive in terms of computation and battery usage for smaller resource constrained devices. Also, the time to discover new sensors in the environment is inherently influenced by the period with which the metadata is published by the sensor component. Nevertheless, even with these considerations in mind it is considered that the decoupled nature provided by the approach overcomes the disadvantages that it may entail.

5.2.3.4 Context Model

For the stigmergic model to work entities must have a common representation of the context information used to describe the environment. It is possible to use a variety of methods depending on the modelling techniques used to represent the context, or on how extensible the approach needs to be.

Current research [151] indicates there are a number methods pervasive computing systems can use to model context. Key-value models provide one of the simplest and most frequently used methods. Context values are defined as a series of attribute-value tuples that describe different facets of the environment. The work done by Schilit et al. [138] is a prime example of such a model in use. Graphical models such as those of Henricksen et al. [66] use UML

style diagrams to model context in pervasive computing systems. Context models based on ontologies [152, 163, 32] have also proven to be a powerful and extensible means of modelling context within pervasive computing.

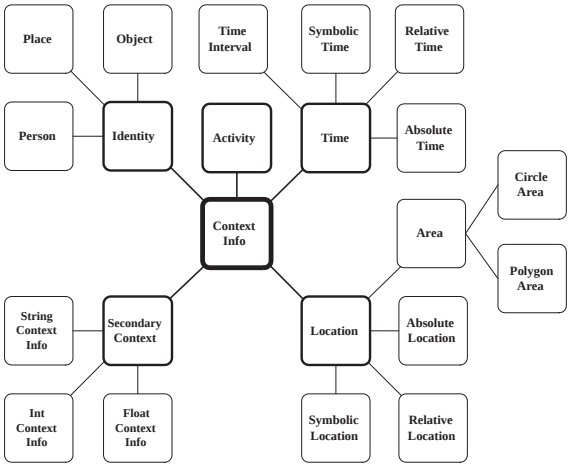


Figure 5.7: Model of context information used in Cocoa framework.

However, for the Cocoa framework it was decided to use an approach based on a key-value model and structure it around the Dey et al. [42] definition of context. In this definition, location, identity, time, and activity are the primary pieces of context used to characterise a situation. These pieces of context form the first level of the context model. All of the other types of context are located on the second level. These can be described as secondary context and are used to highlight other forms of information such as the light intensity of a room or the email address of a person. Figure 5.7 provides an overview of the context model used in the Cocoa framework. Each piece of context in the model is represented as an attribute-value tuple within the framework. For example, the location of a toaster might be represented as *"location = kitchen"*. To describe the environment fully the framework uses collections of these key-value pairings to characterise the different parts of the environment. An illustrative example is provided later in this section of how this is achieved.

To provide a more extensible means of expressing the different forms of context in the framework the context model also captures the different types of location, identity, time, and

secondary context an illustration of which can be seen in figure 5.7. In this instance the identity of the different parts of the environment can be expressed via their type - place, person, or object - and also through their actual identity. The code extract in listing 5.5 shows how the different types of identity context are declared in the framework.

```
1 //Creating an object identity with the id Telephone.
2 Identity idObj = new ObjectIdentity("Telephone");
3
4 //Creating an person identity with the id Telephone.
5 Identity idPer = new PersonIdentity("Peter");
6
7 //Creating an object identity with the id Telephone.
8 Identity idPla = new PlaceIdentity("O'Reilly Institute, F32");
```

Listing 5.5: Declaring identity context information using Cocoa.

Time can be defined as either a symbolic, relative, or absolute expression of time. It is also possible to specify time as an interval or period of time. In the framework a symbolic time represents an abstract notion of time. For example, time can be expressed as *"lunch time"*, *"bed time"*, or "tea *time*". A relative value for time defines a specific point in time in relation to another distinct point in time. For example, 60 minutes after dinner time or 2 hours before 14:00 on the 16 July 2005. Absolute time is defined in terms of Coordinated Universal Time (UTC). An interval of a time is a length of time marked off by two distinct points in time representing the start and end of the interval. Examples of how symbolic, relative, absolute, or an interval of time are defined in the framework can be seen in listing 5.6.

Location information can be defined as both a discrete point or as a bounded area. In the framework a discrete point can be expressed in terms of an absolute, relative, or symbolic location. While a bound area can be defined as a circle, polygon, or symbolic location. An absolute location is defined as a GPS latitude/longitude coordinate in the framework. A relative location is expressed in relation to some other object, space, or location. For example, a toaster could be 5 meters east and 10 meters north of the kettle. A circular area is first specified by a discrete point that acts the center of the circle and a radius that defines the extent to which the area extends to. A polygon area consists of a discrete geological point

```
1 //Creating an absolute time context.
2 Time timeAbs = new AbsoluteTime(System.currentTimeMillis());
3
4 //Creating a relative time context a 3600000 msec after timeAbs.
5 Time timeRel = new RelativeTime(timeAbs,3600000);
6
7 //Creating a symbolic time context Tea Time.
8 Time timeSym = new SymbolicTime("Tea Time");
9
10 //Creating an interval context starting at timeRel and lasting 20 sec.
11 Time timeInt = new TimeInterval(timeRel, 20000);
```

Listing 5.6: Declaring time context information using Cocoa.

that acts as the point of origin for the area and a series x-y values that determine the extent of the area. A symbolic location represents an abstract view of a location. It can represent both a discrete geological point or a bounded area. For example, John is located at *"Pete's desk"* or the phone is located in the room *"O'Reilly Institute, F32"*. Listing 5.7 provides an example of how each of these types of locations are defined within the framework.

```
1 //Creating absolute location context as a GPS coordinate.
2 Location locationGPS = new GPSCoordinates("53°20'38\"N,6°4'58\"W");
3
4 //Creating a relative location context 5 meters east and
5 //10 meter north of locationGPS.
6 Location locationRel = new RelativeLocation(locationGPS, 5, 10);
7
8 //Creating a symbolic location at O'Reilly Institute, F32.
9 Location locationSym = new SymbolicLocation("O'Reilly Institute,F32");
10
11 //Creating a circular area context with a radius of 10 meters
12 //and center location at locationRel.
13 Location locationCir = new CircleArea(locationRel, 10);
14
15 //Creating a polygon area context that defines a 20 by 30 meter area.
16 double[] xpoints = {-10,10,10,-10};
17 double[] ypoints = {20,20,-15,-15};
18 Location locationPol = new PolygonArea(locationGPS,xpoints,ypoints);
```

Listing 5.7: Declaring location context information using Cocoa.

In the current version of the context model, activity context information is only expressed

in one form. The code excerpt in listing 5.8 provides an illustration of how the activity context is defined within the framework.

```
1 //Creating activity context with the value set to idle.
2 Activity activity = new Activity("Idle");
```

Listing 5.8: Declaring activity information using Cocoa.

Secondary context is also represented in the attribute value manner. Where the pairings determine the the type of context and its value. In the framework these values can be expressed as either a string, integer, or floating point number. Currently, the framework does not restrict the type of context that can be placed in this structure, or specify - other than it being a string, integer or floating point number - the format of the value. Examples of how secondary context in defined in the framework can be seen in listing 5.9.

```
1 //Creating secondary string context with type owner and value pete.
2 SecondaryContextInformation secStr =
3     new StringContextInformation("Owner","Pete");
4
5 //Creating secondary float context with type tempature of value 35.
6 SecondaryContextInformation secFlo =
7     new FloatContextInformation("Tempature",35);
8
9 //Creating secondary integer context of type light intensity with
10 //a value of 300 lux.
11 SecondaryContextInformation secInt =
12     new IntContextInformation("Light Intensity",300);
```

Listing 5.9: Declaring secondary context information using Cocoa.

In describing the environment the framework use collections of context information to characterise the situation for different parts of the environment. This can be seen in the code excerpt shown in listing 5.10 where the pieces of context information declared in listings 5.5, 5.6, 5.7, 5.8, and 5.9 are used to define the context of a *telephone* located at 53°20'38"N, 6°4'58"W at *tea time* being *idle*. It also shows in the secondary context that the *Owner* is *Pete*.

```
1 //Declaring the context for a particular part of the environment.
2 Collection scCollection = new HashSet();
3 scCollection.add(secStr);
4
5 Context context = new Context(idObj,activity,
6                                locationGPS,timeSym,
7                                scCollection);
```

Listing 5.10: Declaring context information in Cocoa.

In taking this approach we have chosen to provide the framework with a context model that is well-defined and simply constructed. The aim has been to provide entities of all resource capabilities with a context model that they can use to describe the environment and to reduce the computation requirement for interpreting the context information that is implicitly shared between entities. However, in achieving this the trade-off has been to limit the scope to which entities can reliability use the model to describe the environment. In this model the primary context information is well understood and defined, however, the secondary context information can be open to interpretation as key-value pairings have no agreed meaning. To avoid this limitation an alternative approach could be to use a context model based in ontologies. This would provide an extensible approach that could describe the environment in greater detail. However, it would typically require more resources than the context model described above and so it was felt the extensibility and scope of the model needed to be limited in favour of being able to run on the resource constrained devices normally found in pervasive computing environments.

5.2.4 YABS

The scripting component, YABS (Yet Another Behavioral Script), provides a prototypical implementation of the scripting language described in chapter 4. Its function in the framework is to interpret scripts and to generate the intermediate objects used by the stigmergy runtime environment to control the behavior of individual entities. The rational for providing a separate component for defining entity behavior, is firstly, to encapsulate the functionality of interpreting the script within one component, and secondly, to ensure that it is possible to

load alternative implementations of this component into the framework without the need for major revisions of the framework. The approach gives the framework flexiblity and the ability to be extended to include new functionaility. For example, it may be possible at a later stage to do an implementation based on a graphical interface or using a XML.

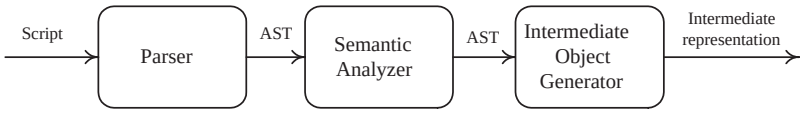


Figure 5.8: Structure of scripting component

5.2.4.1 Scripting Component

A script is initially loaded into the framework when an entity is first initialised. Figure 5.8 shows the phases used to interpret the script. The first phase loads and parses the script from a file and completes a lexical and syntax analysis of the script. It also ensures that any inherited scripts are also loaded and that an analysis of them is completed for the the next phase of the interpreter. The parser produces an abstract syntax tree of the script which is passed to the semantic analyser. The semantic analyser checks the script for semantic errors and also gathers type information for the last phase of the interpreter. The last phase generates an intermediate object representation of the script. This consists of a series of objects that represent the initial proximity to be used by the entity, references to the behaviors the entity can perform, and the mappings to be used to regulate the entity's behavior. In other words, they provide the initial object representations of the three primitives - L , B and M - defined in the stigmergic model in chapter 3. Together, these objects act to provide the stigmergy runtime environment with the parameters it requires to determine the behavior of the individual entities.

Figure 5.9 shows the main parts of the scripting component. The *CompilerComponent* is an abstract class providing the basic functionality required for the component to be integrated in the Cocoa framework. It inherits from *AbstactCompiler* which provides methods for accessing the intermediate objects generated by the scripting component. The concrete

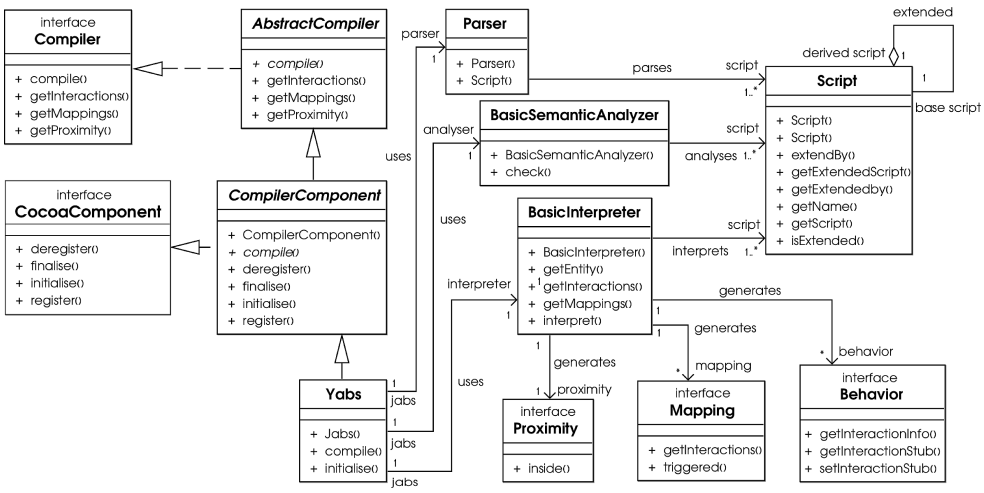


Figure 5.9: Class Diagram of the YABS component.

implementation of *CompilerComponent* is provided by *YABS*. *YABS* uses *Parser*, *BasicSemanticAnalyzer*, and *BasicInterpreter* to generate the intermediate objects required by the stigmergy runtime environment. Each of these provide the functionality required for the different phases used in the interpreter.

The current implementation replies on JavaCC [103] to generate the parser used to read the script. JavaCC is a Java compiler compiler. A tool similar to lex or yacc, it reads a grammar specification to produce a parser capable of recognising matches in the grammar. The parser shown in figure 5.9 - *Parser* - is the result of JavaCC reading a specification that defines the grammar of the scripting language described in chapter 4. The *Parser* is responsible for the first phase of the interpreter. It loads the script - *Script* - from file and performs a lexical and syntax analysis generating an abstract syntax tree of the script. All dependences for the script are also parsed in preparation for the next phase.

The *BasicSemanticAnalyzer* is used to perform the second phase of the interpreter. It takes the abstract syntax tree produced by the parser and checks it for semantic errors. The analyser produces warnings if variables are not declared, if declared variables are assigned the incorrect value, or it an entity behavior does not exist. The *BasicSemanticAnalyzer* also extracts the type information for the last phase of the interpreter. This phase is done by

BasicInterpreter. It generates the intermediate objects *Proximity*, *Mapping*, and *Behavior* that are used by the stigmergy runtime environment to determine the behavior of individual entities.

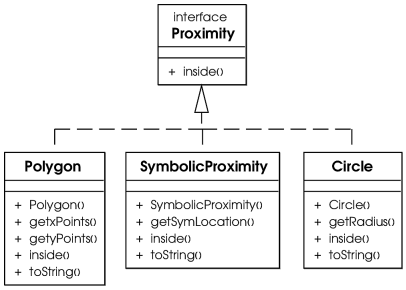


Figure 5.10: Class Diagram of the intermediate objects used for proximity.

5.2.4.2 Intermediate Objects used for Proximity

L, the proximity function, is used by the stigmergic model to determine the local environment for an entity. Within the scripting language the proximity function can be defined as either a radius, polygon, or symbolic area around an entity. This can be seen in section 4.3.2. The initial object representation of the proximity function can be observed in the class diagram shown in figure 5.10. These are generated by the scripting component, YABS, and are used by the stigmergy runtime environment to initialise L, the proximity function, for the entity. *Circle* is used by the framework to define a circular proximity that extends to a predefined radius around the entity. Any part of the environment contained within that region is classified as being in the entity's local environment by the framework. The entity is located at the center of the circle. *Polygon* defines a polygon proximity for an entity. It takes three or more pairs of coordinates, specified in the script, to define the polygon. The entity's current location acts as the reference point for the polygon. An example of how the polygon proximity is defined in a script can be seen in listing 4.2. *SymbolicProximity* is used by the framework to define a symbolic area around the entity. This type of proximity is used by the entity when there is a strong definable boundary, such as room, or building.

5.2.4.3 Intermediate Objects used for Behaviors

B, the behavioral set defines the set of possible behaviors the entity can perform. These are declared in a script as a list of references to specific implementations of the entity's behaviors. An example of how the behaviors are declared can be see in listing 4.3. The behaviors are not directly implemented by the script or the Cocoa framework. The actual implementation of the behaviors is done by application developers following a particular API defined by the framework. This API can be seen in figure 5.11. The collection of behaviors produced by the scripting component are passed to the stigmergy runtime environment which uses them to initialise B, the behavioral set, for the entity.

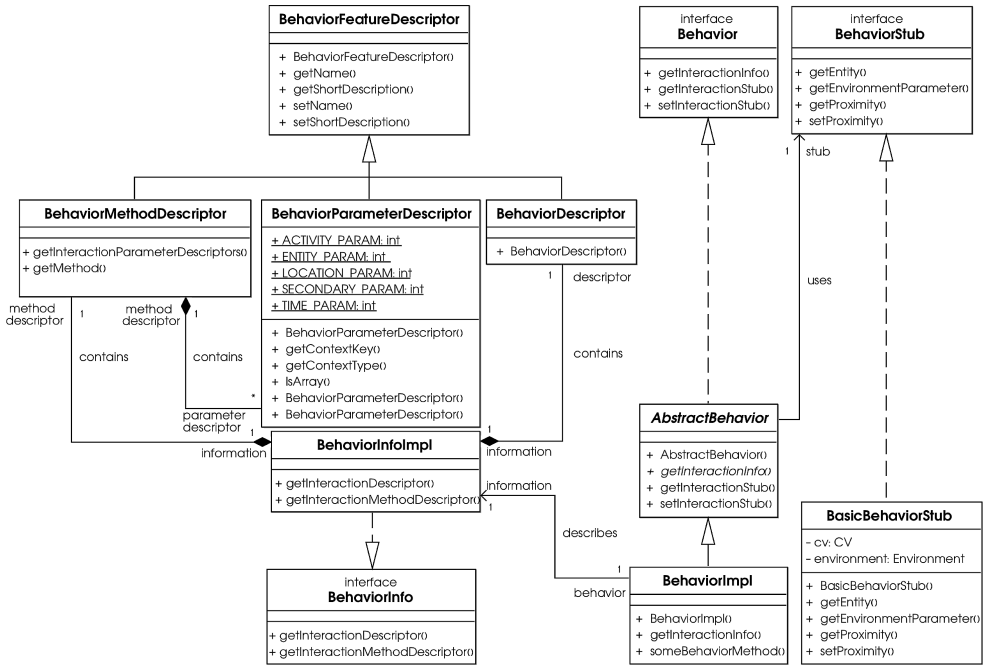


Figure 5.11: Class Diagram of the intermediate objects used for behaviors.

Figure 5.11 shows how the behaviors are represented within the framework. *AbstractBehavior* is an abstract class providing methods for connecting into the framework. In this case for accessing the *BehaviorStub*. The *BehaviorStub* is set by the framework and provides the behaviors with methods for retrieving environmental parameters and for dynamically re-

defining the entity's proximity function. *BehaviorImpl* is the concrete implementation of a behavior. It instructs those actuators used in performing the behavior how to manipulate the local environment for the entity. The implementation of this object is not provided by the framework but by application developers. *BehaviorImpl* also uses *BehaviorInfoImpl* to describe to the framework how to use the behavior. It contains *BehaviorDescriptor* which provides the name of the behavior and a brief textual description of it. It also contains *BehaviorMethodDescriptor* which holds a series of *BehaviorParameterDescriptors*. Together these two objects are used by the framework to describe the method that is to be called by the framework and the type of parameters the framework needs to pass to the behavior. In this case, the method in *BehaviorImpl* is *someBehaviorMethod,* which does not require any parameters for it to be invoked by the framework. It should be noted that *BehaviorInfoImpl* must also be implemented by the application developer when developing the behavior.

The information provided by *BehaviorInfoImpl* is used both by the scripting component and the stigmergy runtime. The scripting component uses the information to determine if the right number and type of parameters are being passed to the behavior. The stigmergy runtime relies on the information to calculate the right parameters to pass to the behavior when it needs to trigger it for the entity. The types of parameters that can be passed by the stigmergy runtime are limited to the different types of context information in the context model described in section 5.2.3.4. How and what parameters are passed by the stigmergy runtime are discussed later in section 5.2.5.3.

The rational for structuring the behaviors for entities in this way, is first, to ensure that the specfic behaviors are able to access the framework, and secondly, to provide a flexible approach that gives developers an expressive means of defining the behaviors for entities while still allowing the framework to trigger them. However, this implies that developers have to work at the interface level of the actuators, which is at times onerous. This may not be such an issue as it is believed that the behaviors will be developed by competent developers and continuously reused by application developers through the scripts.

5.2.4.4 Intermediate Objects used for Mappings

The M function maps the entity's contextual view onto the behavioral set B. The function is defined in a script as a series of mappings that use predicates of context information that are true when matched with information in the entity's current contextual view. On obtaining a full match the mapping is considered to be completed and the mapped behaviors can be triggered by the framework. The script uses a mapping based on temporal intervals. The scripting component generates a series of these mappings that are passed to the stigmergy runtime environment. The stigmergy runtime environment uses them to initialise the M function for the entity.

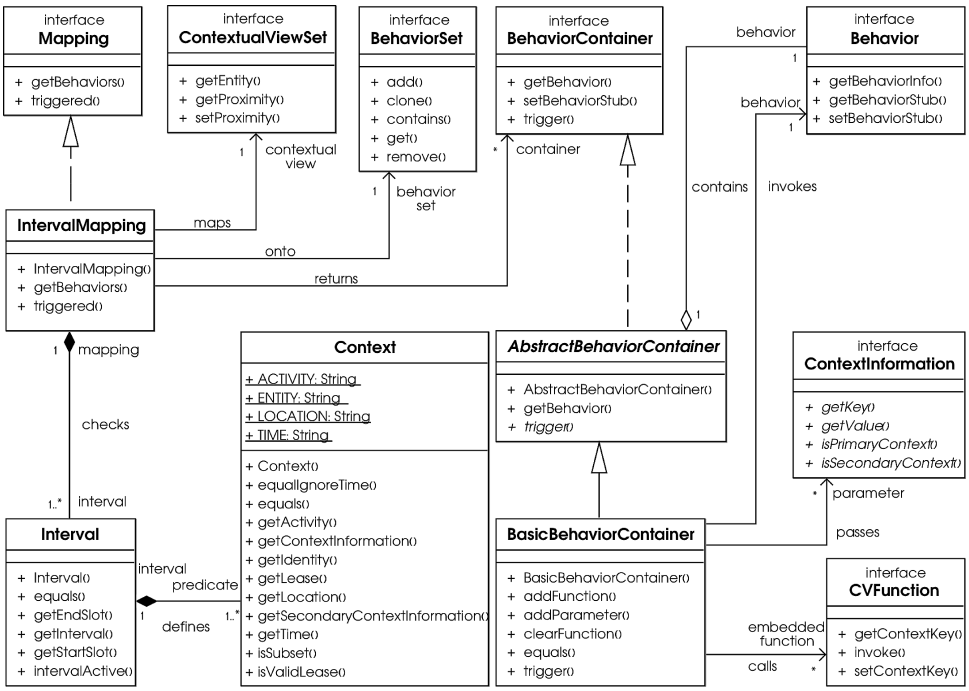


Figure 5.12: Class Diagram of the intermediate objects used for mapping.

The initial object representation of the mappings can be observed in the class diagram shown in figure 5.12. Each of the mapping statements in a script are represented as an *IntervalMapping*. It uses *Interval* to specify the different intervals within the mapping and their relationship to one another. An *Interval* is defined by a series of contextual predicates.

When matched with the entity's current contextual view the interval is considered to be active. *IntervalMapping* periodically checks *ContextualViewSet* to determine if *Intervals* in the mapping are active. When the intervals are found to be active in the correct temporal sequence *IntervalMapping* can complete the mapping to the *BehavioralSet*. A more detailed description of the mechanisms used are presented later in section 5.2.5.

The script also defines the parameters to be used to invoke the behavior when the mapping has been successfully completed. The parameters for the behavior can either be specified directly with a piece of context information or defined through the use of embedded functions. Section 4.5 describes in more detail how the scripting language tailors the behavior of entities through the passing of context information and the use of embedded functions. In the framework the parameters defined in the script and the embedded functions to be used when determining the other parameters for the behavior are contained within a concrete implementation of *BehaviorContainer*. In combination with the stigmergy runtime the container manages the invocation of the behavior by calling the necessary embedded functions - *CVFunction* - and determining the parameters - *ContextInformation* - to be passed to the *Behavior*. The container is initialised by the scripting component and passed, with the mappings, to the stigmergy runtime environment. They are returned by *IntervalMapping* to the framework when the mapping from the entities contextual view to its behavioral set has been satisfied.

5.2.5 Stigmergy Runtime

The stigmergy runtime controls the behavior of an entity. It is responsible for managing the entity's contextual view and for triggering the stigmergic responses of the entity. It provides an implementation of the three primitives - L, B and M - as described in section 3.5 and the runtime environment for managing the behavior of the entity as described in section 3.4.

Each cycle of the runtime environment is composed of three stages as illustrated in figure 5.13. The first stage retrieves the context information from the environment and updates the entity's contextual view, $C_{V_{e_n}}(t)$. Context information, $C_{e_n}(t)$, is included within the contextual view when L, the proximity function, applied to the context information returns

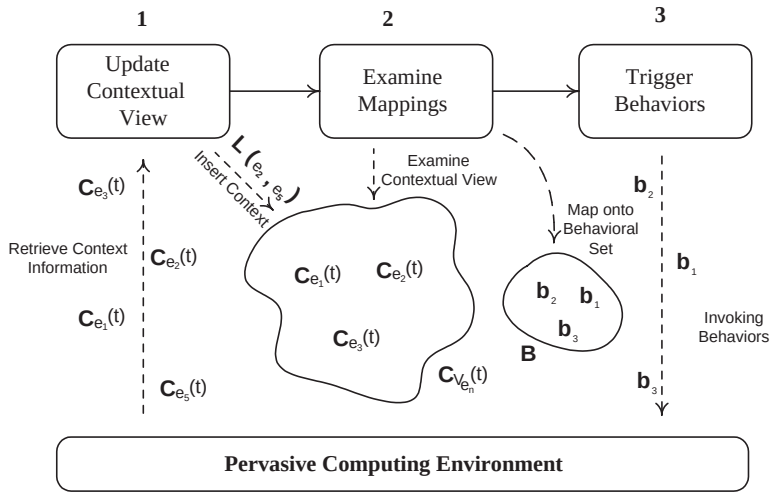


Figure 5.13: Stages used in runtime environment.

true. This indicates that the context information in question is related to the proximity specified by the entity.

The second stage implements the M function. It consists of a series of mappings between the entity's contextual view and the behavioral set, B. The stage operates over a number of cycles gathering information from the entity's contextual view. At each cycle it propagates the state of the mappings and determines if one has been triggered.

The final stage is responsible for invoking the behaviors associated with any of the triggered mappings. It takes the implementation of the behavior and passes the parameters indicated by the script to it. This may include values from the script or context information that needs to be derived from the entity's current contextual view. Once the parameters have been determined the behavior can be invoked by the runtime environment.

5.2.5.1 Observing the Local Environment

The first step for the runtime environment is to determine the local environment for the entity and to provide a representation that it can use to adapt the behavior of the entity. This requires that the runtime be able to retrieve the context information from the local

environment. It achieves this by using the context channel of the communication driver. The information from the channel is gained through a process of cooperation with other entities in the vicinity who openly share the context information that they have derived from the environment. The approach ensures the framework can build up an entire picture of the local environment with minimal amount of effort. It also maintains the separation between the entities through the decouple communication model used to propagate the context information.

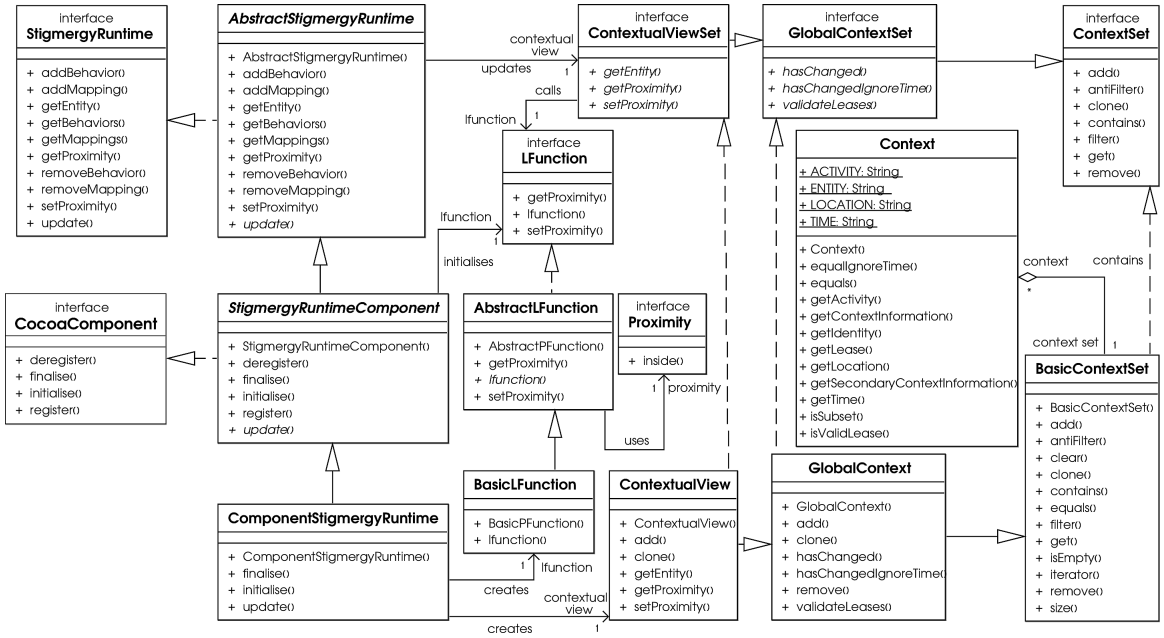


Figure 5.14: Class Diagram of the main parts of stigmergic runtime use to observer the local environment.

Figure 5.14 shows the main parts of the stigmergy runtime environment used to observe the local environment for the entity. The entity's contextual view, $C_{V_{e_n}}(t)$, is implemented by *ContextualView*. It provides the stigmergy runtime environment with a representation of the entity's local environment. Context information retrieved from the context channel is added to *ContextualView* when *LFunction*, an implementation of the L function, returns true. The framework relies on the intermediate object *Proximity* to initialise the L function within the stigmergy runtime environment.

At each cycle of the runtime environment the context information retrieved from the communication driver is inserted into the contextual view when within the correct proximity of the entity. At the same stage any information that is not related to the current vicinity of the entity is removed. The contextual view is also purged of context information that is no longer valid. This is determined by the leasing mechanism outlined in section 5.2.3.1.

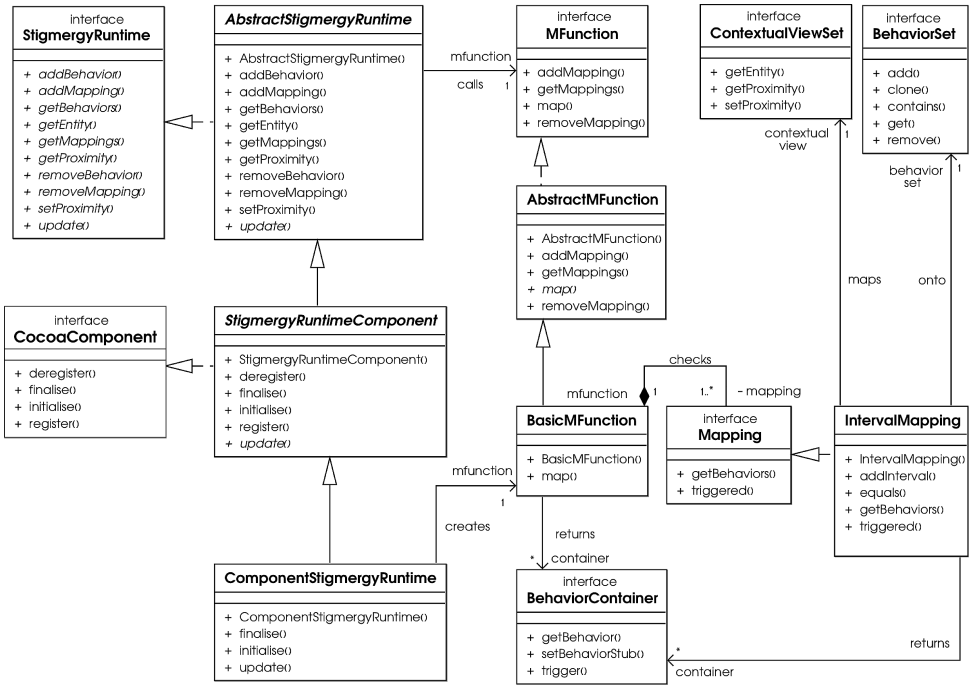


Figure 5.15: Class Diagram of main parts of stigmergy runtime used in the mapping function.

5.2.5.2 Examining the Local Environment

Figure 5.15 shows the main parts of the stigmergy runtime environment that are used to map the entity's current view of the environment onto the behaviors it can exhibit. *BasicMFunction* provides the concrete implementation of the M function for the runtime. It inherits from *AbstractMFunction* which implements the basic parts the *MFunction* interface. The interface is called periodically by *StigmergyRuntimeComponent* to determine if the stimuli present in the local environment effect the behavior of the entity. *BasicMFunction* consist of a series of

Mappings. When the *MFunction* is called the mappings are checked to determine if they have been triggered. The function returns those behaviors that have been mapped. Currently only one type of mapping has been implemented by the framework. This is the *IntervalMapping* described earlier in section 5.2.4.4.

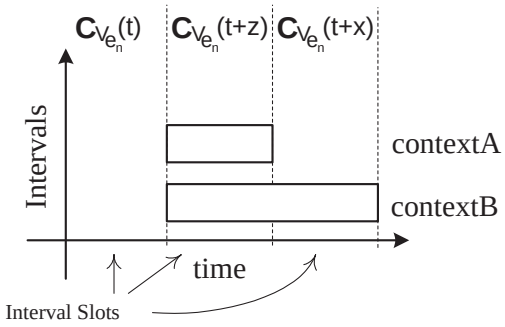


Figure 5.16: Interval slots.

The stigmergy runtime monitors the state of the mappings in the second stage of the runtime environment. The stage operates over a number of cycles, gathering information on when each interval starts and finishes. Over time the relationships between the intervals are built up and it is at this point that it is possible to determine whether the observations satisfy the constraints declared in the mappings. In the runtime environment this process starts by first taking the interval relationships defined in the mapping and dividing them into slots as illustrated in the example shown in figure 5.16.

The process starts at the first slot, where the entity's current contextual view is used to determine if the next slot is valid. A slot is valid if the intervals defined in that slot are active and the remaining intervals are found not to be active. If this is the case then it is possible to move to that slot. To remain in this slot it must be valid in all subsequent contextual views, unless the next slot is valid in which case we move to that slot. If the slot is not valid then the process starts again at the first slot. On reaching the last slot the relationships between the intervals have been satisfied and the mapping can be triggered for the mapped behaviors, at which stage the whole process starts again.

5.2.5.3 Manipulating the Environment

The indirect communication mechanisms use in the stigmergic model are mediated through modifications of the local environment. The model achieves this through entities altering their behavior to physically change the environment in some way. The behaviors use the different actuators in the environment to perform the behavior. The actuators allows the entity to physical change the state of the environment. In the stigmergic model the actuators required to implement the behavior are all connected to the entity. The approach ensures entities can respond quickly to a rapidly changing environment, while also, promoting their autonomy as self-contained units with the ability to modify the local environment.

In the framework the scripting components, YABS, provides the stigmergy runtime environment with the behaviors that the entity can use to manipulate the state the environment. Section 5.2.4.3 has already outlined how the behaviors are implemented in the framework. The behaviors defined in a script are used to initialise stigmergic runtime's behavioral set, B. When the M function maps the entity's contextual view onto the behavioral set it is the responsibility of stigmergy runtime environment to invoke the behaviors associated with any of the triggered mappings. The mapped behaviors are returned by the *MFunction* to the runtime enviuronment in *BehaviorContainers*. The container, as previously described in section 5.2.4.4, contains the behavior and the parameters the behavior requires to be invoked by the framework. The parameters have either been defined in the script or need to be determined through the use of embedded functions that have been specified in the script. To invoke the behavior the stigmergy runtime environment uses the *BehaviorContainer* to determine the parameters and to pass them to the behavior when it is triggered. The invocation of the behavior by the framework alters the behavior of the entity which in turn changes the state of the pervasive computing environment creating a stigmergic response by the entity.

As described in chapter 4 the scripting language defines a number of approaches that can be used to specify the parameters to be passed to a behavior. The initial method passes context information declared in the script. Listing 4.25 is a prime example of this approach in use. In essence the defined context information is static and can be held in the *BehaviorContainer* until it needs to be passed to the behavior. The second, more dynamic approach, uses the

context predicates defined in the script to filter a subset of the context information held in the entities current contextual view. An example of this approach in a script can be seen in listing 4.27. The *BehaviorContainer* holds the predicate until the behavior needs to be invoked. At this stage it is used to determine the subset of the context information to use from the entities current contextual view. The framework then uses the behavior descriptors, defined in section 5.2.4.3, to determine the type of context information to pass to the behavior. If at that stage more than one piece of context information is left, one is randomly selected to pass as a parameter to the behavior.

The last approach uses the embedded functions - *maximum*, *minimum*, *random*, *minority*, *majority*, *average* - to determine the parameters to be passed to the behavior. An example of the use of these functions can be seen in listing 4.27. In the framework each of these functions implements the *CVFunction* interface as described in section 5.2.4.4. The *BehaviorContainer* holds references to the functions to be called for each of the parameters for the behavior. Again, the behavior descriptors are used to determine the type of context information required by the behavior. The function then iterates over the entity's current contextual view to determine the maximum, minimum, random, minority, majority, or average value for that particular type of context information. The function returns the value that can then be passed to the behavior as a parameter.

5.3 Possible Configurations

In providing a prototypical implementation of the stigmergic model and the scripting language the Cocoa framework has aimed to develop an extensible and flexible approach. To achieve this the framework uses a modular design where each of the components uses well defined interfaces. This can be seen in the previous section where interfaces are used to define each of the main components. By taking such an approach it allows alternative implementations of these components to be developed without the need to redevelop the whole framework. The framework is also constructed in such a way that it allows different components to be loaded at runtime depending on the entity and the environmental configurations. It also provides a means of applying different configurations with little overhead.

5.3.1 Cocoa Configuration File

The different component configurations for an entity are specified through a configuration file that is read when the framework is first initialised. An example of of such a file can be seen in appendix B. It is formated in the same way as a Java property file, in that, it contains a list of key/value pairings. These are loaded into an environment configuration component which acts as a repository for the configuration. The component provides the rest of the framework with access to the configuration. A reference to the component is retrieved in the same way as a reference to the other components. Listing 5.11 illustrates how this is achieved. Figure 5.17 shows the class diagram of this component.

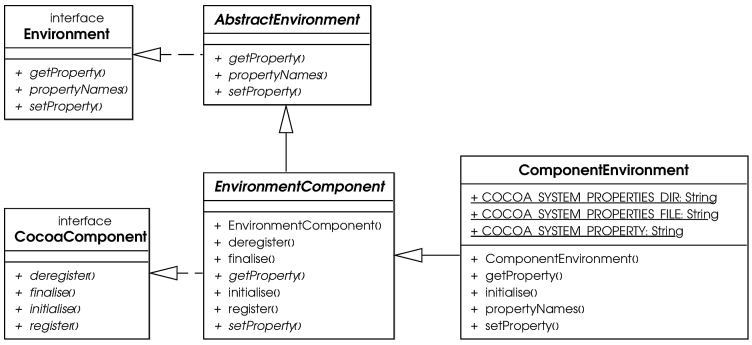


Figure 5.17: Class Diagram of the environment configuration component.

The configuration file contains a list of the components the framework is to load and the configuration for each of these components. Once the configuration is loaded into the environment configuration component, the framework can start to load the individual components into the framework and initialise each of them.

```
1 //Error code omitted to simplify presentation.
2 //Retrieving environment configuration component.
3 Environment environment;
4 environment=(Environment)Cocoa.components.retrieve(Cocoa.ENVIRONMENT);
```

Listing 5.11: Retrieving reference to environment configuration component.

5.3.2 Loading Components

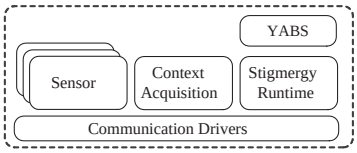
Each of the main components described in section 5.2 implement the *CocoaComponent* interface. The interface defines four methods - *register*, *initialise*, *finalise*, and *deregister* - that allow components to be integrated into the Cocoa framework. The *register* method announces the component to the framework. The *initialise* method allows the framework to set up the component within the framework. The *finalise* method stops the component. The *deregister* method removes the component from the framework. When the framework has retrieved the list of components to be loaded it creates an instance of each one and calls the register and initialise methods on each the components. When the entity is shutting down the framework calls the *finalise* and *deregister* methods on each of the components that are loaded.

5.3.3 Component Configurations

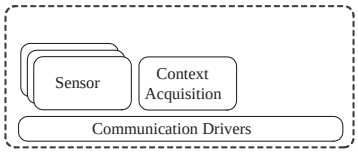
By taking this approach it is possible to start the framework with a variety of different component configurations. This gives great flexibility in the choice of the configurations that can be used by the entity allowing it to tailor its configuration to suit both the resources available to it and other environmental considerations such as the network. The modular structure of the framework also ensures that new functionality can easily be included into the framework. This can be achieved either by adding a new component or by providing an alternative implementation of an existing component.

It is possible to arrange the components implemented in section 5.2 in seven different ways. This does not include the various filters that can be inserted into the sensor and context information channels of the communication driver. It also does not include the different implementations of the main components, for example, the various sensor components or whether the communication driver is based on STEAM or on Siena. Figure 5.18 provides an overview of the configurations that can be achieved. The component configuration shown in figure 5.18(a) is the main configuration promoted in chapter 3 for entities roaming through a pervasive computing environment. It is expected that this will form the base configuration for entities.

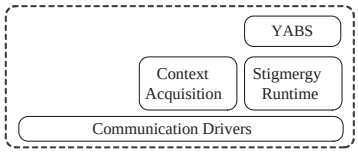
The other configurations may be of use when configuring entities for different environmen-



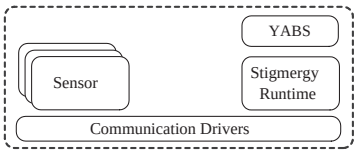
(a) A typical configuration for most entities. It consists of one or more sensors, a context acquisition component, and a stigmergy runtime component in combination with the scripting component, YABS. Underlying these components are the communication drivers.



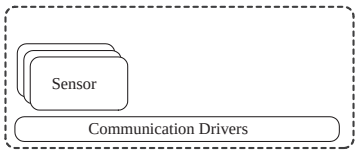
(b) A configuration consisting of one or more sensors, a context acquisition component. In this configuration the cocoa component produces context information that is disseminated to other entities through the communication drivers.



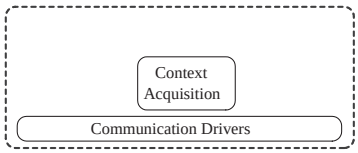
(c) The configuration uses a context acquisition component in combination with a stigmery runtime component and a scripting component. Communication drivers are used to receive sensor data and context information.



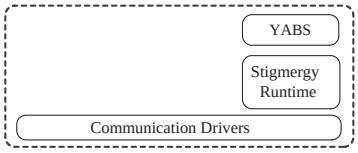
(d) In this configuration one or more sensors are used with a stigmergy runtime and scripting component. The communication drivers disseminate the sensor data and retrieve context information for the stigmergy runtime.



(e) The configuration consists of one or more sensor components. The communcation driver is used to disseminate the sensor data produced by the sensors.



(f) The configuration contains a context acquisition component. It uses the communication drivers to obtain sensor data and to disseminate context information to other entities.



(g) The configuration consists of a stigmergy runtime component and scripting component. The communication drivers are used to obtain context information for the stigmergy runtime.

Figure 5.18: Possible component configurations of Cocoa.

tal parameters. For instance, the configuration shown in figure 5.18(g) could be used by an entity short of computational resources. It receives context information from other sources than its own. Whatever the merits of each of the configurations the possibility of achieving them allows us to experiment with them to find the optimal configuration for the framework.

5.4 Entity Development

The previous sections have described in detail how the stigmergic model and the scripting language have been implemented in the Cocoa framework. This section looks at how the framework is applied to developing entities. As a of means illustration an example entity is used to demonstrate the main steps required in creating an entity with the Cocoa framework. The entity is based on an augmented desklight you might find on any *smart* office desk. The section starts with an overview of the steps required to develop an entity and then continues with a detailed description of the process.

5.4.1 Overview of Entity Development

Developing an entity with the Cocoa framework starts by identifying the entity and understanding the behaviors they can perform within the environment. In the case of the desklight the behaviors are based on its ability to turn the light on or off. The effect on the environment is either to increase or decrease the amount light within the surrounding area. Once the entity and its behaviors have been identified the next step is to use the framework to integrate the entity into the pervasive computing environment. A process that is covered by the following steps:

1. Identify the sensors that the entity will use to sense the physical environment. Using the sensor abstraction described in section 5.2.2.1 develop the components for integrating these sensors into the framework.

2. Using the abstraction described in section 5.2.3.1 create a context acquisition model that the entity can use to derive context information from environmental sensors.

3. Implement the behaviors of the entity using the abstraction defined in section 5.2.4.3.

4. Write a script to define how the entity is to behave within the pervasive computing environment.

5. Write the configuration file, as described in section 5.3.1, for the entity.

For the majority of cases the process of developing an entity can be reduced to the last two steps or even to just the last step. This depends greatly on how the entity is configured and whether the sensors, context acquisition model, behaviors, or script have already been written for other entities. Sensor components only need to be written once for each type of sensor. It can then be reused by other entities using that type of sensor. This also applies to the context acquisition models which can be reused between different entities. It is also possible to reuse the implemented behaviors between entities of a similar type, as can the script. The last step is the only part of the development process that has to be done for each entity.

5.4.2 Creating a Sensor Component

One of the first steps in developing an entity is to implement the components for integrating the sensors into the framework. This is achieved by providing a concrete implementation of the sensor abstraction described in section 5.2.2.1. A specific implementation of this abstraction needs to developed for each type of sensor if an entity is to be able to use it. Each of the implementations provides the functionality for connecting to the sensor and for retrieving readings from the sensor. The component can then produce the sensor data and metadata that the entity requires to sense their local environment. Once the sensor component has been implemented for the sensor it can be reused by other entities using the same type of sensor. For the desklight to be able to detect the state of the local environment it needs to use sensors to determine such physical attributes as light intensity or location.

5.4.3 Developing the Context Acquisition Model

The next step in developing an entity is to provide a context acquisition model that can be used to derive context information from the environmental sensors. The development of

the model relies on a concrete implementation of the *Model* and the *ModelDriver* interfaces that are described in sections 5.2.3.2 and 5.2.3.1. The implementation of *ModelDriver* is used to incorporate the model in to the framework, while the implementation of the *Model* interface harnesses a particular technique to acquire context information that will suit the environmental configuration and entity. In the case the of desklight entity the model is based on simple IF-THEN rules, though other techniques can be used to acquire the context information for the entity such as Bayesian networks.

```
1 //Error code omitted to simplify presentation.
2 public class DeskLightOnBehavior extends AbstractBehavior{
3   //Information on behavior.
4   private BehaviorInfo info;
5   public DeskLightOnBehavior(String name){
6     super(name);
7     this.info = new DeskLightOffBehaviorInfo(name);
8   }
9   //Method used to tigger entity behavior
10  public boolean on(){
11    //Code to turn on light
12    return true;
13  }
14  public BehaviorInfo getInteractionInfo(){
15    return this.info;
16  }
17 }
```

Listing 5.12: Code for on behavior of desklight entity.

5.4.4 Implementing Behaviors for an Entity

Once the sensor components and the context acquisition model have been developed for the entity it is then necessary to implement each of behaviors for the entity. This is achieved by following the API described in section 5.2.4.3. For each behavior a concrete implementation of *AbstractBehavior* and *BehaviorInfo* need to be provided. The sample code shown in listing 5.12 is an example of a concrete implementation of *AbstractBehavior*. In this case, *DeskLightOnBehavior* implements the *on* behavior of the desklight. The behavior is triggered by the framework by calling the *on* method. It should be noted that in this case the framework

does not have to pass any parameters to trigger this particular behavior. *DeskLightOnBehavior* also implements the methods for accessing the objects that describe the behavior to the framework.

```
1  //Error code omitted to simplify presentation.
2  public class DeskLightOnBehaviorInfo implements BehaviorInfo{
3    private BehaviorDescriptor behaviorDesc;
4    private BehaviorMethodDescriptor behaviorMethod;
5    public DeskLightOnBehaviorInfo(String behaviorName){
6      Class behaviorClass = Class.forName("ie.tcd.DeskLightOnBehavior");
7      Method method = behaviorClass.getDeclaredMethod("on", null);
8      behaviorMethod = new BehaviorMethodDescriptor(method, null);
9      behaviorDesc = new BehaviorDescriptor();
10     behaviorDesc.setName(behaviorName);
11   }
12   //Provides a description of the method to be called and
13   //the parameters to be passed.
14   public behaviorMethodDescriptor getBehaviorMethodDescriptor(){
15     return behaviorMethod;
16   }
17   //Provides a brief description of the behavior.
18   public BehaviorDescriptor getBehaviorDescriptor() {
19     return behaviorDesc;
20   }
21 }
```

Listing 5.13: Sample code for behavior information of on behavior desklight entity.

The sample code shown in listing 5.13 provides a concrete implementation of *BehaviorInfo* for the *on* behavior of the desklight entity. It describes to the framework the method that will trigger the behavior and the parameters that need to be passed to trigger the behavior. It also provides a brief description of the behavior.

5.4.5 Define Entity Behavior

At this stage of developing an entity it is necessary to define how it will behave in a pervasive computing environment. This is done by writing a script for the entity. For the desklight entity to turn on the light when a person is nearby and the room is dark and turn it off when nobody is close, the script may appear as that shown in listing 5.14. In this instance the L

```
desklight extends object{
  proximity(5)  //Set L function to a radius of 5 meters.
  //Declare on and off behaviors of the entity.
  behavior on = "ie.tcd.DeskLightOnBehavior"
  behavior off = "ie.tcd.DeskLightOffBehavior"

  context SomePerson  //Declaring a context predicate SomePerson.
  SomePerson.person = any
  context SomePlace  //Declaring a context predicate SomePlace.
  SomePlace.place = any
  SomePlace.light = dark
  context LightOn  //Declaring a context predicate LightOn.
  LightOn.object =  this
  LightOn.activity = "on"
  context LightOff   //Declaring a context predicate LightOff.
  LightOff.object = this
  LightOff.activity = "off"

  //Define mapping to turn desklight on
  map [LightOff, SomePlace][LightOff, SomePlace, SomePerson] onto {
    on()
  }
  //Define mapping to turn desklight off
  map [LightOn, SomePerson][LightOn] onto {
    off()
  }
}
```

Listing 5.14: Defining entity behavior for desklight entity.

function is set to a 5 meter radius around the desklight entity. The behavioral set B contains two behaviors for the entity. The on behavior that is defined in the previous section and the off behavior which turn the light off. The M function that maps the entity's contextual view onto its behavior set is defined through the declared context predicates and the mapping statements.

5.4.6 Instantiating the Entity

The last step in creating an entity with the Cocoa framework is to write the configuration file that tells the framework what components to load and how to initialise them. This is the only step in the development cycle that needs to be done for each entity. For a particular

```
1 //Components to load into framework
2 cocoa.components=ie.tcd.cs.dsg.comms.steam.SteamContextComms:ie.tcd.cs
  .dsg.comms.steam.SteamSensorComms:ie.tcd.cs.dsg.cocoa.
  contextacquisition.ComponentContextAcquisition:ie.tcd.cs.dsg.yabs.
  compiler.Yabs:ie.tcd.cs.dsg.cocoa.runtime.ComponentStigmergyRuntime:ie
  .tcd.cs.dsg.sensors.GPSSensor
3 //Parameters to intialise context acquisition component
4 cocoa.model.drivers=ie.tcd.SimpleModelDriver
5 cocoa.model.name=simplemodel
6 cocoa.model.lease=10000
7 //Static context
8 cocoa.entity.object=pete's desklight
9 cocoa.entity.color=red
10 //Parameters to intialise scripting component.
11 cocoa.yabs.path=/opt/scripts
12 cocoa.yabs.script=desklight
13 //Parameters to intialise STEAM communication driver.
14 steam.mode=mobile
15 steam.proximity=50
16 //Parameters to intialise GPS sensor component.
17 gpsensor.data.lease=10000
18 gpsensor.metadata.lease=60000
```

Listing 5.15: Configuration file for desklight entity.

instance of the desklight the configuration file for the entity may look as that shown in listing 5.15. A full version of this configuration file can be seen in appendix B. The configuration is the same as that shown in figure 5.18(a) where the entity uses a GPS sensor, context acquisition component, a scripting, and stigmergy runtime environment. The entity also uses the STEAM communication driver to disseminate and receive sensor data and the context information. The configuration file also contains the parameters for initialising the individual components. It also holds the static context information of the entity which is, in this case, the name of the entity and color of the desklight.

5.5 Summary

This chapter presents a prototypical implementation of the stigmergic model and of the scripting language used to define entity behavior. The current prototype is called Cocoa has been

implemented on Linux using Java. The chapter began, in section 5.1, with an architectural overview of the framework describing the main components and their functionality within the framework. Section 5.2 continued with a more detailed description of the main components and the algorithms used to control the behavior of the individual entities. The chapter then progressed to section 5.3 where the possible configurations of the framework were described. This section also discussed how the framework is configured for each of the entity in a pervasive computing environment. The last section demonstrated how individual entities are developed with the Cocoa framework using a desk-light as an example. Overall, in building the framework, the intention has been to provide a flexible platform that would allow experimentation with the stigmergic model. The use of a modular design has helped to facilitate this goal.

Chapter 6

Experimental Evaluation

The chapter describes the evaluation of the approach proposed by this thesis for supporting the development of pervasive computing environments. In conducting the evaluation the objective has been to determine whether the stigmergic model defined in chapter 3 and the prototype implementation described in chapter 5 are adequate to build the type of pervasive computing environments envisioned in section 3.1.1. In order to ascertain if this is true a selected number of application scenarios from a range of domains were used to determine whether this approach can realistically meet the criteria for developing these types of environments. Specifically, the chapter presents a number of evaluation experiments, which have been conducted using these scenarios, to establish whether the requirements defined in chapter 3 have been met.

The evaluation began by investigating the methods used in previous research projects as described in section 6.1. Using the review that this provides as a guide section 6.2 defines a set of criteria that are used in combination with the requirements defined in chapter 3 to determine whether the objectives of the thesis have been met. In assessing these section 6.3 presents a set of application scenarios, describing each application scenario, detailing the resulting implementation using the Cocoa framework, and evaluating the effectiveness of the framework in meeting the demands of the criteria. The evaluation employs both real world implementations of application scenarios as well as implementations that were assessed within a simulated environment. Section 6.4 summaries the overall effectiveness of the approach proposed in the thesis in meeting the criteria.

6.1 Evaluating Pervasive Computing Systems

Weiser observed in [168] that "*applications are of course the whole point of ubiquitous computing*", indicating that to evaluate such systems it is necessary to build and experiment with them to determine their success in meeting the vision Weiser set for ubiquitous computing. For the most part the evaluation of ubiquitous computing systems have taken this form, in that, they are validated through their application. As a result, the evaluation of ubiquitous computing systems are generally of a qualitative nature rather than a quantitative one.

6.1.1 Approaches used by Previous Projects

Of the projects reviewed in chapter 2 the majority have used the above method of evaluation. For example, in the Aura project [56] Garlan et al. deployed their system across a university campus and have developed a set of applications to use it. Using these applications they demonstrated how Aura manages the tasks of users in a pervasive computing environment.

In Ambiente's cooperative project Streitz et al. developed the concept of Cooperative Buildings [153]. These were flexible and dynamic environments that provided cooperative workspaces supporting communication and collaboration between users. To test the feasibility of their approach they developed the iLand environment [154] which incorporated a number computer-augmented objects, Roomware [153], that provided information and support for group and individual interaction within the environment.

The Stanford Interactive Workspace Project [78] have created a meeting room, called iRoom, and developed a large number of applications to demonstrate how the collaborative work of users can be supported. They also evaluated their initial goals for the project; providing a robust platform that is both extensible and portable [77, 126]. To demonstrate portability they ported the iROS system to multiple platforms - UNIX, Windows, Mac and WinCE - and languages - Java, C++, and Python. To show application extensibility they used an application scenario - SmartPresenter - to demonstrate how the different levels of indirection used into the iROS system facilitates the extensibility of the environment. To illustrate robustness they performed a series of experiments to demonstrate how the recovery time of the EventHeap after failure was not noticeable to the participants and therefore

acceptable. They also demonstrated how the decoupled communication mechanism used in the EventHeap could temporally mask transient failure of entities.

In the One.World project [60, 61, 62] Grimm et al. used four criteria - completeness, complexity, performance, and utility - with which to evaluate the One.World architecture. The completeness criterion addressed whether the One.World architecture was sufficiently powerful and extensible enough to support the development of useful applications. The complexity criterion was used to determine if it was hard to develop applications with One.World architecture. The performance criterion evaluated the ability of the One.World architecture to perform under normal workloads. The utility criterion questioned whether the One.World architecture would enable others to be successful. To assess the criteria Grimm et al. first developed a number of applications to illustrate the completeness criterion. To evaluate the complexity criterion they furthered analysed the implementations to determine if their approach simplified the development of the applications. In evaluating the performance criterion they measured the the scalability of a number of the key services in the architecture. To determine utility Grimm et al. conducted an experimental comparison of One.World with a number of other distributed systems technologies using students. The students developed applications using these technologies and were later interviewed about the different software engineering issues they faced in developing the applications.

6.1.2 Evaluation Frameworks for Pervasive Computing

As Schmidt points out [141] the evaluation of pervasive computing systems is still not fully understood, in that there does not exist a common set of criteria against which these systems can be evaluated. In some part this can be seen in the above projects in the different approaches they have taken to evaluate their research. As a consequence there are difficulties in comparing results rigorously and quantitatively. The reasons for this are generally attributed to the relative novelty of the research field. However, there have been some attempts in recent times to develop a consensus on the criteria to use to evaluate such systems.

Scholtz et al. [144] have developed a set of evaluation areas for user evaluations of pervasive computing applications. The areas of evaluation proposed by Scholtz et al. include: attention,

adoption, trust, conceptual models, interaction, invisibility, impact and side effects, appeal, and application robustness. The attention criterion determines how distracting devices are to users. It uses metrics such as the number of times a user must change focus, or the overhead of switching among foci. The adoption criterion evaluates the willingness of users to use the application. This criterion uses metrics such as the rate of use, or the value users place on the technology. The trust criterion determines the user's belief in a system not misusing their personal data. This criterion uses metrics to determine the type of information the user is required to divulge and the amount of control they have over that information. The conceptual model criterion examines the users understanding of the device or program. It uses metrics to determine the degree of match between the user model and the behavior of application, or awareness of application capabilities. The interaction criterion determines how well users and systems work together. This criterion uses metrics to ascertain the effectiveness of completing a task, or the ability of the system to support collaborative interaction of users. The invisibility criterion investigates the ability of the system to integrate into the user's environment. Metrics such as a user's control of the system and the accuracy of the system's contextual model are used to evaluate the invisibility. The impact and effect criterion determines the contribution the system makes in supporting the user. This criterion uses metrics based on the changes in user productivity, or the ability of the system to modify user behavior. The appeal criterion examines how attractive the application is to the user and how it adds to the users enjoyment and quality of life. Metrics such as enjoyment level, or ratings of application look and feel are used to evaluate the appeal of the application. The application robustness evaluates the ability of a pervasive computing system to handle faults. This criterion uses metrics such as the percentage of transient faults that were invisible to the user, or the time from user interaction to feedback. As can be observed the evaluation framework proposed by Scholtz et al. is primarily concerned with user-centered measures rather than evaluating the effectiveness of the framework in developing pervasive computing applications.

An alternative approach is that proposed by Ranganathan et al. [129]. They have developed a set of metrics that can be divided into three categories - system, programmability, and usability - for evaluating the effectiveness of a pervasive computing system. System support

examines the areas of context sensitivity, security and discovery. For context sensitivity the quality of the sensed or derived context information is evaluated along with the effectiveness of its use in a pervasive computing system. For security the expressiveness of defining security polices, the ability of users to control access to their private information, and the unobtrusiveness of the security mechanisms used are evaluated. For discovery, the ability of the system to find resources that meet certain requirements is assessed. In the area of programmability they evaluate a system's ability to support multi-device adaptation and partitioning, to support application mobility, to aid application and service composition, to support the use of context information in adapting behavior and to automate the configuration of the system. They also propose the use of more traditional metrics such as man-hours and lines of code in evaluating the programming framework and toolkits used to develop pervasive computing applications. The area of usability assesses a pervasive computing system under a number of metrics. These include the number of head turns as an indicator of user distraction, the physical movement of a user that is not a direct part of the user's task and the need for a user to have a prior knowledge of the system to be able to use it. While the metrics proposed by Ranganathan et al. do not provide an exhaustive approach to evaluating pervasive computing systems they do capture some of the main issues in their evaluation.

6.1.3 Simulating the Physical Environment

The evaluation frameworks described above typically require an implementation that can be deployed in a real world scenario for it to be evaluated. However, while there can be no real substitute to actually building a system and allowing users to evaluate its effectiveness in meeting the defined criteria, a number of projects [10, 105, 114] have been investigating the use of simulators to simulate the physical environment. This method allows researchers to gain an initial evaluation of the usefulness of their approach before having to actually deploy the system in a real world scenario.

The reasoning used for employing such a method of evaluation is generally attributed to the considerable effort that is required to develop systems that can be used in real world evaluations. The implementation of the system needs to be of sufficient standard not to

effect the experience of the users during the evaluation. This generally requires extra time in development and is often made more difficult with hardware that is not well supported or is less than reliable. For instance, obtaining consistent readings from sensors or developing with small embedded devices. Also, the expense incurred in developing complete systems for such environments is often prohibitive.

The use of a simulator provides a means of overcoming these difficulties. Simulators allow researchers to control and adjust different environmental parameters that may be difficult to achieve in the real world. Simulators also allow researchers to reproduce results which are often more difficult to obtain in real world scenarios due to the increased volatility of the environment. It is also possible for researchers to observe how different types of hardware can be combined, or to even experiment with new kinds of hardware before having to develop it.

6.2 Proposed Criteria for Evaluation

The thesis has promoted the use of stigmergy as the basis for the development of a framework that supports a decentralised approach to organising the components of a pervasive computing environment. Using the work described in section 6.1 as a guide this section proposes a set of criteria that can be used to evaluate this approach. The overall aim of the evaluation is to determine whether the use of stigmergy can provide a solution capable of building the type of pervasive computing environments envisioned in section 3.1.1 and which are equivalent to, if not better than, those environments using more centralised techniques. As a consequence, the proposed criteria are primarily focused on evaluating the middleware aspects of building such environments rather than other concerns. Accordingly, the thesis relies on three criteria and corresponding questions to evaluate the approach:

C1: Completeness. Is it possible to build pervasive computing environments using the principle of stigmergy as the basis of the underlying infrastructure? The criterion, based on Grimm et al's evaluation criteria [60, 61, 62], questions whether a framework based on the concept of stigmergy is sufficiently powerful to support the development of a range of useful pervasive computing applications. The importance of completeness is in determining

whether decentralised approaches, such as the one proposed in this thesis, are capable of building pervasive computing environments equivalent to or better than those using more conceptually centralised methods. If completeness can be shown it would illustrate there to be no disadvantage, in terms of the applications that can be built, in using decentralised infrastructures to build such environments.

C2: Coordination. Can collections of autonomous entities coordinate their behavior using the principle of stigmergy to form a coherent pervasive computing environment capable of supporting the activities of users? In proposing this criterion the thesis investigates whether the indirect communication mechanisms of stigmergy can support a decentralised approach to coordinating the behavior of a pervasive computing environment. The choice of coordination reflects a central theme of the thesis and an important consideration for pervasive computing - can meaningful, coherent environments form. It is used primarily to establish if the use of decentralised techniques, such as stigmergy, can coordinate component behavior to form equivalent environments to those using more conceptually centralised infrastructures. If the coordination criterion is shown to hold true, it would indicate there to be no disadvantage in using such decentralised techniques in building pervasive computing environments.

C3: Complexity, programmability. Does separating the complexities of the underlying system with the compositional side of defining entity behavior simplify the development of entities? The criterion is somewhat based on Ranganathan et al.'s criterion of programmability [129] and Grimm et al.'s criterion of complexity [60, 61, 62] in determining how difficult it is to develop entities with the Cocoa framework. It also questions whether the methods used contributed to the rapid development of pervasive computing environments and investigates if the decoupling of entities aids the incremental construction and improvement of solutions. The complexity and programmability criterion was primarily chosen to investigate these aspects and determine if the framework outlined in this thesis provides a suitable approach to building the type of pervasive computing environment envisioned in section 3.1.1.

6.3 Application Scenarios

In proposing the above criteria the objective has been to determine whether the stigmergic model developed in chapter 3 and resulting implementation described in chapter 5 are firstly capable of being used to build useful pervasive computing environments, and secondly, whether they can be constructed in the manner envisioned in section 3.1.1, and lastly, that the programming model encapsulated in the scripting language described in chapter 4 can assist developers in the incremental construction and improvement of these types of pervasive computing environments.

In determining whether these criteria hold true and if the requirements defined in chapter 3 have been met a select number of application scenarios were used to experimentally evaluate the Cocoa framework. The application scenarios chosen were selected for their ability to evaluate the proposed criteria. In all, six application scenarios were used to achieve full coverage of the criteria. The help assistant scenario, in section 6.3.1, in conjunction with the tour guide and the jukebox scenarios, in sections 6.3.2 and 6.3.3 respectively, and to a lesser extent the remaining three scenarios provide a means of testing the first criterion C1 - completeness - as they represent different application scenarios that have previously been used in the field of pervasive computing. To address criterion C2 - coordination - the streetlight scenario, in section 6.3.4, and the voice scenario, in section 6.3.5, were used to evaluate the ability of the Cocoa framework to coordinate components in a pervasive computing environment. Both scenarios require the coordination of multiple components to achieve the desired behavior. The last scenario, presented in section 6.3.6, was used to test the ability of the Cocoa framework to meet criterion C3 - complexity and programmability. The scenario is based around a busy street in the heart of Dublin which provides an idea location for investigating the development and deployment of entities in an urban environment over a period of time.

The following sections provide a description of those application scenarios, outlining the application domain, describing the resulting implementation using the Cocoa framework, and detailing the effectiveness of the framework in meeting the criteria. In evaluating the framework the thesis relies on implementations that can be used in the real world as well as in

simulations. The combination of methods allowed experimentation with a wider range of application scenarios than would otherwise be possible.

6.3.1 Help Assistant

Projects such as Aura [56] and House_n [75, 76] have created pervasive computing applications that assist people in using everyday objects within their environment or in using their environment in a more efficient manner. The assitance is typically provided in a timely manner that suits the users situation. Such an assistant was developed with Cocoa to illustrate the range of applications that can be built with the framework.

6.3.1.1 Scenario

The scenario envisage for a help assistant is one where a user picks up a tool that they rely on to obtain help on how to use a particular object in the environment. Such a scenario may proceed as follows: John arrives into work one day in find a new printer has been installed in the office. Unsure how to use it he picks up his help assistant and brings it over to the new printer. The assistant recognising that help is required, provides John with information on the printer's capabilities. Assistance of this kind could also be provided in other scenarios where help is required by users on the usage of everyday objects within the environment, for instance, the cooker in the home, or the jack used change a flat tyre.

6.3.1.2 Implementation

The implementation of the help assistant scenario required the development of three entities; one to represent the help assistant, another for the printer, and a third for the user requiring help. The construction of these entities followed the development process outlined in section 5.4. The entities were deployed in a simulated environment to allow an initial evaluation of the scenario to be performed. The implementation details of the entities are as follows:

Punter entity. This entity represents the user in the environment who is looking for assistance on how to use the printer or any other everyday object that might be in the

environment. The entity uses a single sensor to determining its location within the simulated environment. The implementation of the sensor component can be seen in appendix C.1. The context acquisition model replies on the data produced by the sensor in combination with the static context defined in the configuration file to derive context information for the punter entity. The context information generated by the model typically comprises of the person's name, location, and time. The implementation of the model is based on IF-THEN rules. For this scenario, no behaviors have been developed for the punter entity. However, even though the framework triggers no behavior for this entity it is still necessary to represent the person requiring help to allow other entities absorb their context. The script for a punter entity as well as a sample configuration file for an entity are shown in appendix C.4. In the simulated environment one punter entity was deployed to represent John in the above scenario.

Printer entity. The printer in the above scenario is represented by this entity. The current implementation is quite basic, only capturing a very limited amount of context information and having no associated behaviors. The reason for such a minimal implementation is that additional functionality for the entity is not required for this scenario. However, it is envisaged that future development of the entity would include behavior relating to the printing of documents. The same location sensor as the punter entity is used. The context acquisition model is based on IF-THEN rules, which typically generates context information that comprises of the printer's name, location, time, and usage details. The script for the printer entity is shown in appendix C.4.1 and the configuration file can be seen in appendix C.4.2. In the simulated environment one printer entity was deployed to represent the printer in the scenario described in section 6.3.1.1.

Help assistant entity. In the scenario John picks up a help assistant tool to find out how to use the new printer in the office. This entity is used to represent the tool that John uses to gain help. In a real world implementation of this scenario it is envisaged that such an entity would run on a PDA device which would allow John to pick it up and receive help. However, in the current implementation a Firefox browser is used to display the information. The same location sensor is used as in the punter and printer entities. The context acquisition model

```
1  public class DisplayBehavior extends AbstractBehavior {
2    private static final String COMMAND_START =
3      "/opt/firefox/mozilla-xremote-client openURL(";
4    private static final String COMMAND_END = ",new-tab)";
5    private BehaviorInfo info;
6    public DisplayBehavior(String behaviorName) {
7      super(interactionName);
8      this.info = new DisplayBehaviorInfo(behaviorName);
9    }
10   public BehaviorInfo getInteractionInfo() {return this.info;}
11   public boolean display(SecondaryContextInformation url) {
12     URL urlObj = new URL((String)url.getValue());
13     Runtime runtime = Runtime.getRuntime();
14     Process proc = runtime.exec(COMMAND_START + urlObj.toString() +
15                                 COMMAND_END);
16     return true;
17   }
18 }
```

Listing 6.1: Code for display behavior of help assistant entity.

generates context information relating to the name of the entity, its location, and current time. Its implementation is based on IF-THEN rules. One behavior has been implemented for the help assistant entity, called *display*, it opens a web page on the Firefox browser. The Java implementation for this behavior can be seen in listing 6.1.

```
1  assistant extends firefox{
2    proximity(5)  //Set L function to a radius of 5 meters.
3    context SomePerson  //Declaring a context predicate SomePerson.
4    SomePerson.person = any
5    context SomeObject  //Declaring a context predicate SomeObject.
6    SomeObject.object = any
7    SomeObject.usage = any
8    map [SomePerson, SomeObject] onto { //Define mapping.
9      display(SomeObject.usage)
10   }
11 }
```

Listing 6.2: Script for help assistant entity.

The *display* behavior is triggered when someone is nearby and when information on the object - printer - is available to display. The script defining this behavior for the help assistant

entity can be seen in listing 6.2. The help assistant script extends the firefox script. The firefox script, which is shown in listing 6.3, declares the *display* behavior for the entity and the script is reused in a number of the scenarios described in this chapter. The configuration file for the entity can be seen in appendix C.5. In the simulated environment one help assistant entity was deployed to represent the help assistant in the scenario described in section 6.3.1.1.

```
1 firefox extends object{
2   //Set L function to a radius of 5 meters.
3   proximity(5)
4   //Declaring the display behavior.
5   behavior display = "ie.tcd.DisplayBehavior"
6 }
```

Listing 6.3: The firefox script.

6.3.1.3 Results and Analysis

From running the above scenario in a simulated environment with the deployed entities - punter, printer, and help assistant - it was observed that as John, represented by the punter entity, moved towards the printer with the help assistant that the usage information for the printer was displayed in the Firefox browser. The application scenario, which has been used in other research projects [75, 76, 56], demonstrates how the Cocoa framework can be used to develop similar pervasive computing applications. In doing so, it illustrates the ability of the framework to meet criterion C1 - completeness.

The help assistant scenario also helps to demonstrate requirement R1 - physical integration. The high-level abstractions provided in the scripting language allow entities in the scenario to sense and interact with the physical environment without the difficulties of dealing with low-level sensors or actuators in the scenario. The framework also provides the low-level abstractions required to integrate sensors and actuators into the environment. In the above scenario these abstractions can be seen in terms of the location sensor used by all the entities and of the display behavior used by the help assistant tool. Such levels of abstraction ease the development of entities through separating the compositional side of defining entity behavior

with the low level system development of the sensors and actuators. The division provides a logical separation of the concerns making the environment easier to build and maintain.

The scenario also showed the autonomous behavior of the entities, in that, they controlled their own behavior within the simulated environment and operated independently of other entities. Such behavior demonstrates how the Cocoa framework supports requirement R2 - autonomy of components - in a pervasive computing environment.

6.3.2 Tour Guide

Projects such as GUIDE [33, 39], Cyberguide [91], and Interactive Museums [53] have developed pervasive computing applications that provide tourists with information on the locations they visit. A similar type of application was developed to illustrate the range of applications that can be developed with the Cocoa framework.

6.3.2.1 Scenario

Electronic tour guides can take a number of forms depending on the domain and the objective of the guide. In this case, the tour guide was developed to support a scenario such as this one: Jane arrived in Dublin the night before and has awoken in her city center hotel with the intention of exploring the city that morning. She takes her electronic tour guide and heads off. On passing a prominent building she takes out her tour guide to find that the building is the front of Trinity College and the home of the Book of Kells. On getting directions she walks into the college to visit the famous book. Also, during that day the administrator of Trinity College added new tourist information relating to the Douglas Hyde Gallery.

6.3.2.2 Implementation

In many ways the implementation of the tour guide scenario is very similar to the help assistant presented in section 6.3.1, in that, it requires the development of three entities - one for the electronic tour guide, a second entity for the different tourist attractions, and a third for the tourist - which use similar if not the same components. The entities were deployed in a

simulated environment to evaluate the scenario. The implementation details of the entities are as follows:

Punter entity. In this scenario the punter entity represents the tourist, Jane, who is sightseeing in the city of Dublin. The current implementation is the same as that used for the punter entity in the help assistant scenario described in section 6.3.1.2. In the simulated environment one punter entity was deployed to represent Jane.

Tourist attraction entity. The entity represents the different locations or buildings that are of interest to those visiting. The current implementation of the tourist attraction entity is quite simple, only capturing a limited amount of context information and having no associated behaviors. Potentially these entities could capture a richer set of context information and implement a larger range of behaviors, however, for the purpose of the scenario it is not required. The construction of the tourist attraction entity is similar to the printer entity, in that, it uses the same basic components. For example, the location sensor. The script for a tourist attraction entity and a sample configuration file can be is seen in appendix C.6. In the simulated environment a number of these entities are deployed to represented different tourist spots that Jane might visit on her trip to Dublin.

Electronic tour guide entity. This entity is used to represent the electronic tour guide that Jane uses to gain this information. It is expected in a real world implementation that the entity would run on a device such as a PDA allowing Jane to carry it around while sightseeing. However, in the current implementation a Firefox browser is used again to display the information on the different tourist attractions that Jane may visit. The actual construction of the tour guide entity is very similar to the help assistant entity, in that, the basic components are the same. For instance, the location sensor and the display behavior which is used to display information on the Firefox browser. The context acquisition model generates context information relating to the name of the entity, its location, and current time. Its implementation is based on IF-THEN rules. The script, show in listing 6.4, defines the behavior of the electronic tour guide entity. It triggers the *display* behavior when a person is

nearby a tourist attraction where there is information to be displayed. As before, with the help assistant entity, the script of the tour guide entity extends the firefox script shown in listing 6.3. The configuration file for the entity can be seen in appendix C.7. In the simulated environment one electronic tour guide entity was deployed for Jane to use when visiting the different tourist attractions.

```
1 tourguide extends firefox{
2   proximity(100)  //Set L function to a radius of 100 meters.
3
4   context SomePerson  //Declaring a context predicate SomePerson.
5   SomePerson.person = any
6   context SomePlace  //Declaring a context predicate SomePlace.
7   SomePlace.place = any
8   SomePlace.about = any
9
10  //Define mapping.
11  map [SomePerson, SomePlace] onto {
12    display(SomePlace.about)
13  }
14 }
```

Listing 6.4: Script for electronic tour guide entity.

6.3.2.3 Results and Analysis

From running the above scenario in a simulated environment with a punter entity and electronic tour guide entity to represent Jane and the tour guide she uses, plus five tourist attraction entities to represent the different tourist locations she might visit. In the simulated environment it was observed that as Jane moved towards a tourist attraction the electronic tour guide would display information relevant to the attraction.

The tour guide scenario has been used extensively in different research projects [33, 39, 53, 91] to demonstrate the feasibility of various pervasive computing systems. So in a similar fashion to the previous scenario the tour guide scenario also illustrates how the Cocoa framework can be used to successfully build such applications. In doing so, it also demonstrates the ability of the framework to meet criterion C1 - completeness.

From the electronic tour guide it also possible to demonstrate how requirements R1 -

physical integration - and R2 - autonomy of components - are fulfilled by the Cocoa framework. Like the previous scenario of the help assistant the tour guide uses the same high-level abstractions provided by the scripting language to abstract the complexities of dealing with the real world. This can be seen in script used by the electronic tour guide entity shown in listing 6.4. In running the scenario it was also observed that the autonomous nature of the entities, in particular in this case the tourist attraction entities, made it possible to add or remove entities without affecting the operation of other entities in the system. This facilitated the incremental construction of the system, in that, it was always possible to add new tourist attraction entities into the environment.

The extensive reuse of components from the previous scenario - the punter entity, the location sensor component, and the display behavior - also demonstrates how it is possible to achieve the rapid develop of environments through the reuse of entities and components. Such properties successfully demonstrate the ability of the framework to satisfy criterion C3 - complexity and programmability.

6.3.3 The Jukebox

The jukebox is an application that plays different types of music genre depending on who is in its vicinity and to what they like to listen. The jukebox was developed to demonstrate how a pervasive computing application can coordinate its activities with the other entities in the local environment, but also, to show the range of applications that can be developed with the Cocoa framework.

6.3.3.1 Scenario

A typical scenario for the jukebox is centered around a group of people within a room having coffee or talking about the news of the day and having the jukebox playing in the background. Such a scenario for the jukebox could possibly proceed as follows: Thomas is meeting a group of friends for coffee in John's house. On entering John's house the jukebox on the table starts to play some 70s music in recognition of Thomas's preference. However, when Jane arrives the jukebox observes a change in the majority preference to Hip-Hop and starts to play songs

from that particular genre. As the morning coffee comes to an end everyone gets up to leave and as they leave the house the jukebox stops playing.

6.3.3.2 Implementation

The implementation of the jukebox scenario required the development of two entities; one to represent the jukebox, and a second entity to represent those listening to the music in the scenario. The entities were deployed in a simulated environment for experimentation and to gain an initial evaluation of the scenario. The implementation details of the entities are as follows:

Punter entity. The punter entity represents those listening to the music being played by the jukebox. The implementation of this entity reuses the same components, script, and context acquisition model as that used by the punter entity described in section 6.3.1.2. However, an additional piece of context information is specified in the entity's configuration file to define the music preferences of the person. The configuration file for this entity can be seen in appendix C.4.3. In the simulated environment three punter entities were deployed to represent Thomas, John, and Jane in the scenario described in section 6.3.3.1.

The Jukebox entity. This entity represents the jukebox in the scenario. The current implementation of this entity is based on the popular Linux multimedia player, Xmms [158]. To be able to use the player a Java Native Interface was developed to allow the Cocoa framework control the player. The entity uses two sensors one to determine the location of the jukebox in the simulated environment, and a second to ascertain if the jukebox is playing music, is paused, or has stopped playing music. The implementation of the location sensor is the same as that used in previous scenarios (see appendix C.1). For the second sensor the implementation of the component can be found in appendix C.2. The context acquisition model uses the data produced by these sensors to generate the context information for the jukebox entity. The model is implemented using IF-THEN rules that produce context information relating to the state of the jukebox. Two behaviors have been developed for the jukebox entity, called *play* and *stop*, that control how the entity behaves in the pervasive

```
1  //Error code omitted to simplify presentation.
2  public class StopBehavior extends AbstractBehavior {
3    private static final int SESSION = 0;
4    private BehaviorInfo info;
5
6    public StopBehavior(String name) {
7      super(name);
8      info = new StopBehaviorInfo(name);
9    }
10   public boolean Stop() {
11     if (XmmsControl.remote_is_running(SESSION)) {
12       if (XmmsControl.remote_is_paused(SESSION) ||
13           XmmsControl.remote_is_playing(SESSION)) {
14         XmmsControl.remote_stop(SESSION);
15       }
16     }else {
17       return false;
18     }
19     return true;
20   }
21   public BehaviorInfo getInteractionInfo() {
22     return info;
23   }
24 }
```

Listing 6.5: Code for stop behavior of jukebox entity.

computing environment. The *play* behavior takes a single parameter that defines the music genre to be played by the entity. The parameter is used to search the music collection for suitable matches. A single song is then selected from this matched set to be played. The *stop* behavior halts the jukebox. The Java implementation for the *stop* behavior can be seen in listing 6.5.

The script, shown in listing 6.6, is used to define the behavior of the jukebox entity. The *play* behavior is triggered when a person is near the jukebox. The genre of music played depends on what the majority of people prefer to listen to. This is determined by observing the context information from punter entities, in particular the musical preferences of the entities. While this is the current behavior of the jukebox entity it can be modified to react differently to the environment. For instance, it could play different music depending on location, or it could always play what the minority of the people like to listen to. The *stop* behavior is

```
1  jukebox extends object{
2    proximity(10) //Set L function to a radius of 10 meters.
3
4    //Declare behaviors for
5    behavior play = "ie.tcd.PlayBehavior"
6    behavior stop ="ie.tcd.StopBehavior"
7    context SomePerson  //Declaring a context predicate SomePerson.
8    SomePerson.person = any
9    SomePerson.music = any
10   context JukeBoxPlay //Declaring a context predicate JukeBoxPlay.
11   JukeBoxPlay.object = this.object
12   JukeBoxPlay.activity = "play"
13   context JukeBoxStop //Declaring a context predicate JukeBoxStop.
14   JukeBoxStop.object = this.object
15   JukeBoxStop.activity = "stop"
16
17   //Defining mappings.
18   map  [JukeBoxStop]  [JukeBoxStop, SomePerson] onto {
19     play(majority(SomePerson))
20   }
21   map [JukeBoxPlay, SomePerson] [JukeBoxPlay] onto{
22     stop()
23   }
24 }
```

Listing 6.6: The jukebox script.

triggered when there is no one in the vicinity of the jukebox entity. The configuration file for the entity can seen in appendix C.8.1. In the simulated environment one jukebox entity was deployed to play music.

6.3.3.3 Results and Analysis

From running the above scenario in a simulated environment with a jukebox entity and three punter entities representing Thomas, John, and Jane it was observed that the jukebox selected music based on those in its vicinity and would stop playing when there was no one to listen. The jukebox, which is also a scenario used in Gaia [133], reinforces how different types of pervasive computing application can be developed using the Cocoa framework, which again, contributes to showing of the framework meets criterion C1 - completeness. In addition to the requirements - R1 and R2 - shown in previous application scenarios the jukebox scenario

also demonstrates how a pervasive computing application using the Cocoa framework can adapt its behavior to suit the surrounding environment. This can be observed in the way the jukebox entity is able to change the type of music played to suit those listening. In displaying such behavior the scenario illustrates how requirement R9 - adaptability - is satisfied by the framework.

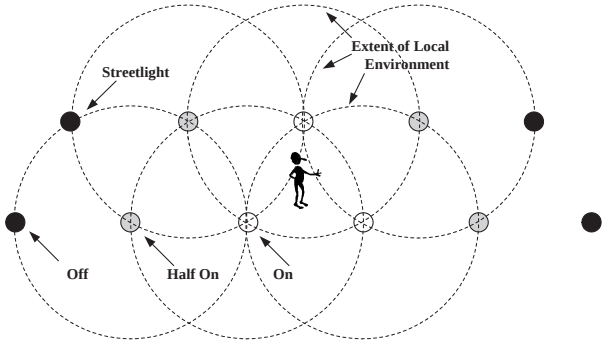


Figure 6.1: An illustration of the streetlight scenario.

6.3.4 Streetlights

The streetlight application scenario was developed to demonstrate how entities of a pervasive computing environment can coordinate their activities without having to directly communicate with each other. It is also used to show how system behavior can emerge from the local interactions of entities. The application is based on a normal set of street lights that you might find along the side of a road where a pedestrian walkway might exist. The lights coordinate their activity, through observing their local environment, to ensure the street is sufficiently lit for the pedestrians to walk safely along the path. Figure 6.1 provides an illustration of the scenario.

6.3.4.1 Scenario

The scenario is located on a street lit by a series of streetlights that extend along the length of the street. By default the lights are off to save energy but turn on in the presence of users, or

half on when the street light beside is fully on. The effect is to have the light follow the person along the street and through the city. A scenario involving such active streetlights could be as follows: Ciara has forgotten to get milk for the morning so puts on her coat and heads down to the local shop around the corner. As she starts to walk down the street the light beside her turns on and light further down the street appears to turn half-on but as she approaches it the light turns fully on and streetlight behind her turns half-on. As she continues to walk the first light is now off and lights in front her continue to light her way to the shop.

6.3.4.2 Implementation

The implementation of the streetlight scenario required the development of two entities; one to represent the streetlight, and another to present a person walking along the street. As before, the construction of these entities followed the development process outlined in section 5.4. To gain an initial evaluation of the scenario the entities were deployed in a simulated environment. The implementation details of the entities are as follows:

Punter entity. In this scenario the punter entity represents the pedestrian, Ciara, going for milk in the shop around the corner. The implementation of this entity is the same as the punter entity described in section 6.3.1.2 for the help assistant scenario and in section 6.3.2.2 for the electronic tour guide scenario. In the simulated environment one entity was deployed to represent Ciara in the scenario described in section 6.3.4.1.

Streetlight entity. Each of the streetlights in the scenario are represented by this entity. The entity uses two sensors one to determine the location of the streetlight within the simulated environment, and a second sensor to ascertain if the streetlight is on, halfon, or off. The implementation of the location sensor is the same as that used in previous scenarios. For the second sensor the implementation is the same as that used in the jukebox scenario in section 6.3.3.2. The context acquisition model uses the data produced from by sensors to derive context information for the streetlight entity. The model is implemented using IF-THEN rules which generate context information relating to the state of the streetlight. Three behaviors have been developed for the streetlight entity, the first one turns the streetlight on, the sec-

```
1 //Error code omitted to simplify presentation.
2 public class StreetLightOnBehavior extends AbstractBehavior{
3   private BehaviorInfo info;
4   public StreetLightOnBehavior(String name) {
5     super(name);
6     info = new StreetLightOnBehaviorInfo(name);
7   }
8   public boolean on() {
9     return StreetLight.getStreetLight().lightOn();
10  }
11  public BehaviorInfo getInteractionInfo() {
12    return info;
13  }
14 }
```

Listing 6.7: Code for on behavior of streetlight entity.

ond one turns the streetlight half-on, and the last one turns the streetlight off. The Java implementation of the on behavior can be seen in listing 6.7. Both *off* and *halfon* behaviors of the streetlight entity have a similar Java implementation to that of the *on* behavior. The script defining the behavior of the streetlight entity is shown in listing 6.8. The *on* behavior is triggered when a person enters the local environment of the streetlight entity. The *halfon* behavior is triggered when other streetlights in the vicinity of the entity are *on*. The *off* behavior of the entity is triggered when the person leaves the local environment and when no other streetlights in the vicinity are on. The configuration file for this entity can be seen in appendix C.9. In the simulated environment ten of these entities were deployed to observe their behavior.

6.3.4.3 Results and Analysis

The scenario described in section 6.3.4.1 was deployed in a simulated environment with a single punter entity to represent Ciara and ten streetlight entities placed in a two lines as illustrated in figure 6.1. From running the scenario it was observed that as Ciara moved along the streetlights they first turned half-on then fully on as Ciara got closer to the streetlight and as she moved away they turned from on to half-on and then off.

While the streetlight scenario is not a typically scenario used by other pervasive computing

```
streetlight extends object{
  proximity(200) //Set L function to a radius of 200 meters.

  //Declaring on, halfon, and off behaviors for streetlight entity.
  behavior on = "ie.tcd.StreetLightOnBehavior"
  behavior halfon = "ie.tcd.StreetLightHalfOnBehavior"
  behavior off = "ie.tcd.objects.StreetLightOffBehavior"

  //Declaring context predicates for streetlight entity.
  context SomePerson //Declaring a context predicate SomePerson.
  SomePerson.person = any
  context SomeLightOn //Declaring a context predicate SomeLightOn.
  SomeLightOn.activity = "On"
  context LightOn //Declaring a context predicate LightOn.
  LightOn.object = this.object
  LightOn.activity = "On"
  context LightOff //Declaring a context predicate LightOff.
  LightOff.object = this.object
  LightOff.activity = "Off"
  context LightHalfOn //Declaring a context predicate LightHalfOn.
  LightHalfOn.object = this.object
  LightHalfOn.activity = "HalfOn"

  //Defining mappings for streetlight entity.
  map [LightOff][LightOff, SomeLightOn] onto {
    halfon()
  }
  map [LightOff][LightOff, SomePerson] onto {
    on()
  }
  map [LightHalfOn, SomeLightOn][LightHalfOn] onto {
    off()
  }
  map [LightHalfOn][LightHalfOn, SomePerson] onto {
    on()
  }
  map [LightOn, SomePerson,SomeLightOn][LightOn,SomeLightOn] onto {
    halfon()
  }
  map [LightOn, SomePerson][LightOn] onto {
    off()
  }
}
```

Listing 6.8: Script for streetlight entity.

projects it does provide another example of how a pervasive computing application can be developed using the decentralised approach encapsulated within the Cocoa framework. In so doing, the scenario also contributes to showing how the framework satisfies the criterion C1 - completeness. The implementation of the scenario also demonstrates how collections of autonomous entities, streetlights in this case, can coordinate their behavior using the principle of stigmergy to form a coherent pervasive computing environment. This can be observed in the manner the street light entities coordinate their individual behavior to ensure the street is sufficiently lit for pedestrians. Such behavior answers the question posed by criterion C2 of whether entities using the principle of stigmergy can coordinate their behavior to the same extent as those using more centralised infrastructures.

The coordination shown also demonstrates how the decentralised coordination of requirement R4 is fulfilled by the framework, while the shared environment used by the entities to communicate illustrates how it is possible to fulfill requirement R3 - spontaneous interoperability - for a pervasive computing environment.

6.3.5 The Voice

The voice is designed to provide groups of people with recommendations for particular web pages or documents. The recommendations are based on what other people are currently viewing in the local environment. Such an application could be used in meetings, or when browsing the internet, or as an added feature to the electronic tourist guide where it could recommend information on different tourist locations. The voice scenario was developed for similar reasons to the street light scenario, in that, it demonstrates how entities of a pervasive computing environment can coordinate their behavior through observing their local environment. In this case, it is used to allow people to coordinate their behavior with others in the same environment. Figure 6.2 provides an illustration of how this is achieved.

6.3.5.1 Scenario

The scenario chosen is based on a meeting in which a number of people are attending to review a collection of academic papers for an up coming conference. The scenario begins with Vinny

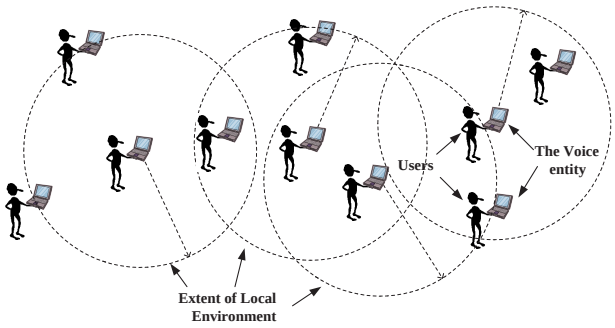


Figure 6.2: The voice scenario.

arriving late: Already 10 minutes late Vinny enters the meeting with his laptop and quietly finds a seat at the table and sits down. He opens his laptop and from the voice application he can see that the meeting has moved onto the second paper after accepting the first paper for the conference. It is also possible for the voice application to be used in other scenarios. For instance, in a tourist scenario a person could determine the best information to look at based on the behavior of those tourists at the tourist spot.

6.3.5.2 Implementation

The implementation of the voice scenario required the development of two entities; one to represent the voice, and another for the person viewing the recommended information. The entities were deployed in an office environment to gain an initial evaluation of the scenario. The implementation details for the entities are as follows:

Punter entity. In this scenario the punter entity represents the person viewing the recommended information. The implementation of this entity is basically the same as in previous scenarios with the exception that a location sensor is not used in preference to statically defining the entity's location in the configuration file. This was necessary as no indoor location service was available at the time of development. The configuration file for this entity can be seen in appendix C.4.4. In the office environment the punter entity runs on a laptop associated with a person. Three punter entities were deployed in the office environment, of

which one represented Vinny.

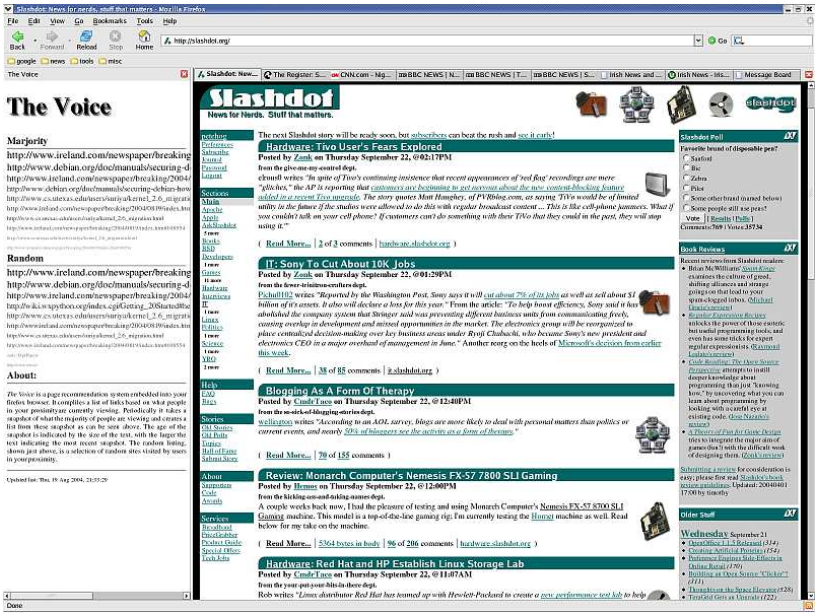


Figure 6.3: User inferface of the voice entity.

The voice entity. In the scenario Vinny uses an application to determine the academic paper being reviewed. This entity is used to represent the application used by Vinny. The current implementation runs as a sidebar in a Firefox browser as can be seen in figure 6.3. The recommendations from the voice entity are placed in the sidebar for users to view. In the office environment the voice entity runs on a laptop. The entity uses a single sensor to determine the page open in the Firefox browser. A location sensor should be used, however, for the same reasons as the punter entity, one is not. Instead, the location of the voice entity is assigned in the configuration file of an entity. The context acquisition model uses the data produced by the page sensor along with the information specified in the configuration file to derive context information for the voice entity. The model is implemented using IF-THEN rules. One behavior of the entity has been implemented for the voice entity, called *recommend*, it updates the sidebar on the Firefox browser with recommendations of documents to view. The script defining the behavior of the voice entity is shown in listing 6.9.

```
1  thevoice extends firefox{
2    proximity(5)  //Set L function to a radius of 5 meters.
3
4    //Declaring recommend behavior for the voice entity.
5    behavior recommend = "ie.tcd.RecommendPageBehavior"
6
7    context SomePerson  //Declaring a context predicate SomePerson.
8    SomePerson.person = any
9
10   context FirefoxBrowser
11   FirefoxBrowser.object = any
12   FirefoxBrowser.page = any
13   FirefoxBrowser.title = any
14
15   map [SomePerson, FirefoxBrowser] onto {
16     recommend(majority(FirefoxBrowser),random(FirefoxBrowser))
17   }
18 }
```

Listing 6.9: Script for the voice entity.

It should be noted that when the *recommend* behavior is triggered two parameters are passed to the behavior. The first parameter specifies what the majority of Firefox browsers in the vicinity of the entity are viewing. The second parameter is a random document being viewed by a browser in the local environment of the entity. The parameters are used to update the sidebar as shown in figure 6.3. The configuration file for this entity can be seen in appendix C.10. In the office environment, three of these entities were deployed to be used by punter entities in the environment.

6.3.5.3 Results and Analysis

From running the scenario, described in section 6.3.5.1, with three punter entities and three voice entities it was observed that the sidebar would change incrementally to make recommendations that reflected the pages being viewed at the time by those using the Firefox browsers. It was found as users click on these recommendations that it reinforced the choice of page being viewed by those in the vicinity causing the page to be indexed higher on the sidebar.

The main objective of the scenario was to evaluate the ability of the framework to coordi-

nate the behavior of autonomous entities in forming coherent environments capable of assisting users. In the scenario such behavior can be seen through the actions of the voice entities who coordinate their individual behavior to provide meaningful recommendations to user. The ability of the voice entities to exhibit such behavior demonstrates how the framework can fulfil criterion C2 - coordination. It also illustrates how the requirement R4 - decentralised coordination - in a pervasive computing environment is supported by the Cocoa framework.

Also, in a similar manner to the jukebox scenario, the voice scenario provides another example of how a pervasive computing application can be developed using the Cocoa framework. This again shows the utility and also the flexibility of the framework in meeting criterion C1 - completeness. It can also be observed that the shared environment, in the form of the context information derived from the physical environment, provides a common interoperation model that supports the spontaneous interaction of entities in the environment. Such a property illustrates how requirement R3 - spontaneous interoperability - is fulfilled by the Cocoa framework.

In addition the voice scenario also demonstrates how a pervasive computing application can adapt its behavior to suit the surrounding environment. This can be seen in the manner the voice entity adapts the content of the sidebar to reflect the activity in the local environment, which again, demonstrates how the framework fulfills requirement R9 - adaptability.

6.3.6 The Westland Row Development

Westland Row is a street located in the heart of Dublin, Ireland. The street is about 250 meters long, and accommodates a number of cafes, newsagents, shops, pubs, and a train station. It is a busy street, with commuters, shoppers, cars, and buses using it on a daily basis. A wireless ad-hoc network has been deployed on Westland Row, with a number of nodes placed along the street. The nodes form a sparse population of wireless network nodes and can be configured to create a variety of network models. The current model uses AODV [119] as the ad-hoc routing protocol for the network and uses a gateway node to access the Internet. The network is part of a project [9] investigating the use ad-hoc networks in urban areas. Westland Row provides both a challenging, and an interesting testing ground

for evaluating pervasive computing applications and for determining the effectiveness of the Cocoa framework to develop and deploy entities in an urban environment. The purpose of using such an environment is to investigate how entities can be developed and deployed in an urban environment over a period of time.

6.3.6.1 Scenario

With the large amount of activity on Westland Row it is possible to run a variety of scenarios with a range of different entities. The following is one example: Peter and Vinny arrange to meet in Westcoast for coffee one morning. Coming from opposite ends of the street they meet at the entrance of the cafe, where they enter and walk up to the counter to order some coffee. The jukebox noticing their entrance starts to play some "trad" music. As they sit down and start to chat the music shifts to more main stream music in reaction to those who have just entered. Another scenario might be: Ciara is shopping along Westland Row to find that the book store is closed. She takes out her shopping assistant to find that it is actually closed for renovation and will open again next week. A similar scenario could be: Ciara is shopping another day on Westland Row and notices some interesting art in the window of a gallery. Wondering who painted it and how much it is, she takes out her shopping assistant to find it is by a local artist and well over her budget.

6.3.6.2 Implementation

The development of Westland Row began by identifying the key entities in the environment, then using the development process highlighted in section 5.4 to implement each of them. The entities identified for the above scenario include an entity to represent a person on the street, entities to act for the different shops located along the street, an entity to operate as the shopping assistant, and an entity to control the behavior of the jukebox in the cafe. Of the entities deployed in Westland Row the majority either ran on laptops, or smaller devices embedded in the environment. The society of entities created were as follows:

Punter entity. Punter was one of the first type of entity to be developed for Westland Row. The entity represents a person on the street, whether they are shopping, having coffee, or commuting to work along the street. The implementation of the entity is based on the punter entities used in previous scenarios. However, the location sensor is different, in that, a GPS device is used to determine the location of the entity. The implementation of the sensor component used for the GPS device can be seen in appendix C.3. It periodically queries a GPS device for changes in location that then are then propagated via a sensor data event. As with the previous implementations of the punter entity no behaviors are implemented for it. The context information generated by the context acquisition component typically comprised of the person's name, location, time, musical preferences. The configuration file for the entity can be seen in appendix C.4.5. In the Westland Row environment the punter entity runs on a mobile device, a laptop, associated with a person. In all three punter entities were deployed in the environment to represent Vinny, Peter, and Ciara in the scenario described in section 6.3.6.1.

Siopa entity. Siopa is the Irish word for *shop* and in the Westland Row pervasive computing environment is the type of entity used to represent the different shops and cafes along the street. The current implementation of the siopa entity is quite basic, only capturing a very limited amount of context information about itself and having no behavior associated with it. It is possible implement a greater range of behavior for the siopa entity and to improve it ability to capture context information but for the purpose of the scenario such an implementation is not required. A sample configuration file for a siopa entity can be seen in appendix C.11, as can the script used for the entity. The siopa entities ran on PC-104s embedded along the street as can be seen in figure 6.4. In all, five siopa entities were deployed to represent the different shops and cafes located along the street.

Shopping assistant entity. Shopping assistant provides information about the different shops in the pervasive computing environment. The current implementation is similar to that of the help assistant and electronic tour guide entities used in previous scenarios. It is based on the Firefox browser which is used to display the information associated with the shops.

(a) Emebedded device used by entities.

(b) Installed Device.

Figure 6.4: One of the embedded devices used in Westland Row development.

The entity only uses one sensor - GPS device - to determine its location. The same the GPS sensor component is used as for the punter entity above. The implementation of context acquisition model is based on IF-THEN rules that generate context information relating to the name of the entity, its location, and current time. One behavior has been implemented for the shopping assistant entity, called *display*, it opens a web page on the Firefox browser. The implementation for the behavior is the same as that used in the help assistant and electronic tour guide entities. The behavior is triggered when someone is nearby and when information is available to display. The script defining this behavior is shown in listing 6.10. A sample configuration file for an entity can be seen appendix C.12. In the Westland Row environment only one shopping assistant was deployed. It ran on a laptop associated with the person shopping.

Jukebox entity. In the scenario presented in section 6.3.6.1 the jukebox begins to play "trad" music as Vinny and Peter enter the cafe. This entity is used to represent the jukebox in the cafe. The current implementation is quite similar to that used for the jukebox in section 6.3.3.2. The only difference is that a GPS device is used instead of the simulated location sensor in the jukebox entity described in section 6.3.3.2. A sample configuration file for a jukebox entity running in the Westland Row environment can be seen in appendix C.8.2. Only one jukebox entity was deployed in the Westland Row environment, which ran on a

```
shoppingassistant extends firefox{
  proximity(100)  //Set L function to a radius of 100 meters.

  context SomePerson  //Declaring a context predicate SomePerson.
  SomePerson.person = any
  context SomeShop  //Declaring a context predicate SomeShop.
  SomePlace.place = any
  SomePlace.deals = any

  //Define mapping.
  map [SomePerson, SomeShop] onto {
    display(SomeShop.deals)
  }
}
```

Listing 6.10: Script for shop assistant entity.

laptop connected to speakers.

6.3.6.3 Results and Analysis

In all there were three punter entities deployed, five siopa's representing some of the shops and cafes on the street, one shopping assistant entity associated with the person shopping, and a jukebox entity in one of the cafes halfway down the street. From running a number of scenarios on Westland Row with the deployed entities the behaviors of the entities were observed. The siopa entities along Westland Row remained passive to changes in their environment, which was as expected due to the current implementation of the siopa entity. The shopping assistant, sometimes carried around by people, would display information about the shops as they walk by. The jukebox entity, with its collection of music, would tailor the selection played depending on the users in it's vicinity. In the scenario above as Peter and Vinny entered the cafe shop the jukebox started to play more folk music to reflect the preferences of the users in the cafe.

The benefit of using stigmergy was observed through the indirect communication mechanisms used by the phenomenon. It was possible to add new entities, remove or upgrade old ones from Westland Row without adversely effecting of rest of the street. This allowed for the environment to be built incrementally and solutions to be improved on over time. Properties which show how the framework meets criterion C3 - complexity and programmability -

and demonstrates requirement R7 - incremental construction. The level of indirectness built into the framework made the overall system less fragile and more stable to disturbances in the environment. The removal of entities from the environment, whether through failure or movement of the entity, did not cause additional problems to the remaining entities. Such behavior illustrates how the Cocoa framework fulfills requirement R6 - robust behavior - for pervasive computing environments. The movement of entities through this urban environment, punter entities in particular, also illustrated how the framework can cope with the mobility of entities and still be able to coordinate the activities of entities. This demonstrates the Cocoa frameworks ability to meet requirement R8 - mobility.

In developing the entities described in the previous section it was also noted that the abstractions provided managed to separate the computational side of acquiring and managing context information with the compositional side of developing pervasive computing applications. The clear separation allowed developers to concentrate on first developing the system-level components and then on implementing the behavior of individual entities. This can particular be seen in the manner the GPS sensor component was developed and incorporated into the entities deployed in Westland Row. The low level abstractions provided by the Cocoa framework allowed the system level development to be performed in isolation to the higher level implementation of entity behavior. The division provides a logical separation of the concerns that makes it easier to build and maintain entities in a pervasive computing environment. For instance, to change the behavior of jukebox to play what the minority of listeners like only requires the alteration of the entity's script and not of any low-level system components.

It was also found that such levels of abstractions allowed for the rapid development of entities as sensors, behaviors, and context acquisition models could all be reused and combined in different ways to develop an entity. For instance, the sensor component for the GPS device was reused in five of the deployed entities, while the display behavior in the shopping assistance is the same as that used in the help assistant and electronic tour guide application scenarios. The extensive reuse of components seen in Westland Row aids the rapid development of such environments and make it significantly easier to develop entities. It was also observed that

the traditional concept of a pervasive computing application shifts somewhat when using Cocoa, as the focus for development is centered on the entity and not solely on any particular application. The applications per say emerge from the pervasive environment as the entities move and reorganise themselves.

The observations answer the question proposed by criterion C3, that separating the complexities of the underlying system with the compositional side of defining entity behavior does simplify the development of entities, while the component architecture of the framework also helps in the rapid development of entities due to the ability to reuse components in different entities.

6.4 Summary

The chapter has evaluated the approach proposed by this thesis for supporting the development of pervasive computing environments. The objective of the evaluation has been to determine whether the stigmergic model defined in chapter 3 and the subsequent prototypical implementation provided in chapter 5 can build the type of pervasive computing environments envisioned in section 3.1.1. Furthermore, whether the proposed approach is capable of building environments equivalent to or better than those using more conceptually centralised infrastructures as those outlined in chapter 2. To ascertain if this is true a selected number of application scenarios, described in section 6.3, were used to demonstrate whether such an approach can realistically meet the criteria, defined in section 6.2, for developing these types of environments. Table 6.1 provides a summary of the different application scenarios used to demonstrate the proposed criteria and to illustrate how requirements described in section 3.1.2 are fulfilled by the framework.

From the table it can be seen that through either a combination of application scenarios or through specific ones that all the criteria outline in section 6.2 have been meet by the prototypical implementation of the stigmergic model. However, two of the requirements were not demonstrated, the first, requirement R5 - scalability, and the second requirement R10 - security and privacy. The most entities deployed in an application scenario was 11 entities in the streetlight scenario. This is not enough to demonstrate the ability of the framework

Application Scenarios	**Criteria**			**Requirements**									
	C1	**C2**	**C3**	**R1**	**R2**	**R3**	**R4**	**R5**	**R6**	**R7**	**R8**	**R9**	**R10**
Help Assistant	○			●	●								
Tour Guide	○		○	●	●								
Jukebox	○			●	●							●	
Streetlights	○	●				●	●						
The Voice	○	●				●	●					●	
Westland Row			●						●	●	●		

=not illustated ○=partially illustrated by ●=illustrated by

Table 6.1: The table contains a summary of the scenario described and the criteria demonstrate and requirements illustrated by them.

to scale. It is, however, believed that as all interactions are mediated through the local environment that it is possible to scale such a system to a large number of entities over a wide area. To comprehensively illustrate this property of the framework and to obtain definitive figures would require the deployment of a large scale system incorporating hundreds of entities within a real world environment. Present levels of resources have not permitted such evaluation at this time. Requirement R10 - security and privacy - has not be implemented in the current prototypical implementation so could be evaluated.

Chapter 7

Conclusions and Future Work

The research presented in this thesis has explored the used of stigmergy in supporting the development of pervasive computing systems. More specifically, it has focused on mimicking the behavior of social insects to construct a society of autonomous entities capable of supporting spontaneous interaction and providing robust system-wide behavior for a pervasive computing environment. This chapter summaries the most significant achievements of the work and outlines its contributions to the state of the art. To conclude, the chapter provides a discussion of related research issues that remain open for future work.

7.1 Achievements

The motivation for the work presented in this thesis arose from the observation that state-of-the-art research in pervasive computing environments has principally focused on providing conceptually centralised infrastructures that coordinate the resources of a specific geographical location. These research efforts, as discussed in chapter 2, are typically designed from the ground up to support the anticipated needs of users, and are usually pre-installed and maintained over the period in which they are in use. However, as pointed out in chapter 3, it is unrealistic to expect all pervasive computing environments to be constructed in this manner. In the future physical spaces are more likely to evolve accidentally into pervasive computing environments as technology is incorporated into the space over the medium-to-long

term. This suggests that pervasive computing environments need to be assembled in a more ad-hoc fashion.

To address this issue the thesis presented a highly decentralised method of organising the components of a pervasive computing environment that supports spontaneous interaction between entities and provides robust system-wide behavior. The inspiration for this work stemmed from the observations made by the French biologist Grassé on how social insects coordinate their actions using indirect communication via the environment, a phenomenon known as stigmergy. The stigmergic approach provided an appealing construct that satisfied the requirements, described in chapter 3, for developing pervasive computing environments. The approach was encapsulated in the stigmergic model, described in chapter 3, that was used to underpin the Cocoa framework for pervasive computing.

The Cocoa prototype implementation of the stigmergic model, is described in chapter 5. Cocoa supports the use of stigmergy to build self-coordinating environments that promote the autonomy of entities. Designed to both support and complement the use of stigmergy, the framework employs a distributed architecture organised in a peer-to-peer fashion. To simplify the implementation and deployment of entities Cocoa supports a programming abstraction encapsulated in a high-level scripting language. Described in chapter 4, the scripting language generalises the methodologies used by social insects to construct a society of autonomous entities capable of responding to the environment in a stigmergic manner.

In summary, the work present in this thesis has focused primarily on investigating techniques based on stigmergy for the design and implementation of pervasive computing environments. Consequently, the main contributions made by this thesis can be summarised as follows:

- An overview of pervasive computing systems with respect to the integration and organisation of devices and applications within these types of the environments.
- A highly decentralized method for organising components of a pervasive computing environment that supports spontaneous interaction between entities and provides robust system-wide behavior.

- A framework, called Cocoa, that supports the use of techniques based on the phenomena of stigmergy to build self-coordinating environments which promote the autonomy of entities.
- A programming abstraction encapsulated in a high-level scripting language that generalises the methodologies used by social insects to construct a society of autonomous entities capable of responding to the environment in a stigmergic manner.
- A demonstration of the Cocoa framework and the scripting languge using a number of application scenarios covering a range of domains.

7.2 Open Research Issues

As is so often the case with research there are some issues that remain open for possible future work. The stigmergic model developed in this thesis is based in the concept of using qualitative stigmergy to mediate the communication of entities in a pervasive computing environment. The use of quantitative stigmergy proved difficult to apply due to the need to deposit artificial pheromones in a virtual environment. However, this type of stigmergy could be of interest and may provide an improved method of controlling the behavior of pervasive computing environments if the problem of managing the virtual environment and the resulting consistency requirements in an ad-hoc environment can be overcome.

In the requirements stated in section 3.1.2, R10 - security and privacy - described the need to maintain the privacy of users and to control access to devices and services within a pervasive computing environment. Section 3.3.8 stated that providing security for pervasive computing environment and ensuring the privacy of users was not addressed directly via the use of stigmergy, though it does not restrict its inclusion at a later stage. The current prototypical implementation of the stigmergic model does not include support for this requirement. To tackle this issue further implementations might consider incorporating a trust-based model, such as in the SECURE project [24], to determine the trustworthiness of users and of entities in a pervasive computing environment.

The current prototypical implementation uses a context model that defines the context

information used by entities to describe the environment. The context model currently uses the concept of primary context information, which is well understood and defined, and secondary context information which consists of key/value pairings. There is an assumption that entities have a common understanding of what these key/value pairings mean, however, there is the potential for them to be open to interpretation. While the current implementation of the context model proved sufficient to prove the viability of the approach presented in this thesis, further work is necessary to ensure the interpretation of context information is consistent between entities. Such an alternative approach may use an ontology to define the context information that can be used in Cocoa.

7.3 Concluding Remarks

This chapter summarised the motivations for and the most significant achievements of the work presented in this thesis. In particular, it outlined how this work contributed to the state of the art in pervasive computing by providing a highly decentralised method, based on the natural phenomena of stigmergy, for organising components of a pervasive computing environment that supports spontaneous interaction between entities and provides robust system-wide behavior. The chapter concluded with some suggestions for possible future work arising from the research undertaken as part of this thesis.

Appendix A

Gammar for Scripting Language

Below is the BNF grammar for the scripting language defined in chapter 4. The grammar was generated using the jjdoc tool of the Javacc [103] utility.

```
1  Script := <IDENTIFIER> ( "extends" Identifier )? ScriptBody <EOF>
2  ScriptBody := Block
3  VariableDeclarator := VariableDeclaratorID ("=" VariableInitializer)?
4  Type := ( "behavior" | "context" | "cvfunction" )
5  VariableDeclaratorID := <IDENTIFIER>
6  VariableInitializer := Expression
7  Expression := ( Assignment | IntervalExpression | ProximityExpression
   | UnaryExpression | PrimaryExpression )
8  UnaryExpression := ( "+" | "-" ) PrimaryExpression
9  Assignment := PrimaryExpression AssignmentOperator Expression
10 ProximityExpression := "proximity" Arguments
11 IntervalExpression := "[" ( NameList )? "]"
12 PrimaryExpression := PrimaryPrefix ( PrimarySuffix )*
13 PrimaryPrefix := ( Literal | ThisExpression | Name )
14 PrimarySuffix := ( Arguments | "." Identifier )
15 ThisExpression := "this"
16 Name := Identifier ( "." Identifier )?
17 NameList := Name ( "," Name )*
18 Identifier := <IDENTIFIER>
19 Literal := ( <INTEGER_LITERAL> | <FLOATING_POINT_LITERAL> | <
   STRING_LITERAL> | AnyLiteral )
20 AnyLiteral := "any"
21 AssignmentOperator := "="
22 Arguments := "(" ( ArgumentList )? ")
23 ArgumentList := Expression ( "," Expression )*
24 Statement := Block | StatementExpression | MappingStatement
25 Block := "{" ( BlockStatement )* "}"
26 BlockStatement := ( LocalVariableDeclaration | Statement )
```

```
27 LocalVariableDeclaration := Type VariableDeclarator ( ","
   VariableDeclarator )*
28 StatementExpression := ( Assignment | ProximityExpression |
   PrimaryExpression )
29 MappingStatement := "map" IntervalExpression ( IntervalExpression )* "
   onto" Block
```

Appendix B

Cocoa Configuration File

The appendix contains a sample configuration file that is used to intialise the Cocoa framework for an entity.

```
1  //Components to load into framework
2  cocoa.components=ie.tcd.cs.dsg.comms.steam.SteamContextComms:ie.tcd.cs
   .dsg.comms.steam.SteamSensorComms:ie.tcd.cs.dsg.cocoa.util.
   SensorMetadataFilter:ie.tcd.cs.dsg.cocoa.util.SymbolicLocationFilter:
   ie.tcd.cs.dsg.cocoa.contextacquisition.ComponentContextAcquisition:ie.
   tcd.cs.dsg.yabs.compiler.Yabs:ie.tcd.cs.dsg.cocoa.runtime.
   ComponentStigmergyRuntime:ie.tcd.cs.dsg.sensors.GPSSensor
3
4  //Parameters to intialise context acquisition component
5  cocoa.model.drivers=ie.tcd.SimpleModelDriver
6  cocoa.model.name=simplemodel   cocoa.model.lease=10000
7
8  //Static context
9  cocoa.entity.object=pete's desklight
10 cocoa.entity.color=red
11
12 //Parameters to intialise scripting component.
13 cocoa.yabs.path=/opt/scripts
14 cocoa.yabs.script=desklight
15
16 //Parameters to intialise STEAM communication driver.
17 steam.mode=mobile
18 steam.range=50
19 steam.period=1
20 steam.proximity=50
21
22 //Parameters to intialise stigmergy runtime
23 cocoa.cv.ignoretime=true
```

```
24
25 //Parameters to intialise GPS sensor component.
26 gpsensor.symlocation=pete's desklight
27 gpsensor.data.lease=10000
28 gpsensor.metadata.lease=60000
29 gpsensor.server=brain gpssensor.port=2947
30
31 //Other parameters for keeping track of symbolic information and
   logging
32 cocoa.staticsymboliclocations=/opt/runtimedata/staticsymboliclocations
33 cocoa.staticsymbolictimes=/opt/runtimedata/staticsymbolictimes
34 cocoa.logger=ie.tcd.cs.dsg.cocoa
```

Appendix C

Application Scenarios

This appendix contains code extracts from the application scenarios presented in chapter 6.

C.1 Location Sensor for Simulated Environment

The source code in this section is of the sensor component used to represent the location sensor in the simulated environment.

```
1  package ie.tcd.cs.dsg.karma.sensors;
2
3  import ie.tcd.cs.dsg.cocoa.cocoa.CocoaException;
4  import ie.tcd.cs.dsg.cocoa.contextmodel.RelativeLocation;
5  import ie.tcd.cs.dsg.cocoa.contextmodel.SymbolicLocation;
6  import ie.tcd.cs.dsg.cocoa.sensor.FormatSensorInformation;
7  import ie.tcd.cs.dsg.cocoa.sensor.IDSensorInformation;
8  import ie.tcd.cs.dsg.cocoa.sensor.LocationSensorInformation;
9  import ie.tcd.cs.dsg.cocoa.sensor.Metadata;
10 import ie.tcd.cs.dsg.cocoa.sensor.SensorComponent;
11 import ie.tcd.cs.dsg.cocoa.sensor.Data;
12 import ie.tcd.cs.dsg.cocoa.sensor.SensorException;
13 import ie.tcd.cs.dsg.cocoa.sensor.StringSensorValue;
14 import ie.tcd.cs.dsg.cocoa.sensor.SymbolicCoverage;
15 import ie.tcd.cs.dsg.cocoa.sensor.TypeSensorInformation;
16 import ie.tcd.cs.dsg.cocoa.util.MetadataBroadcastThread;
17
18 import java.awt.Window;
19 import java.awt.event.ComponentEvent;
20 import java.awt.event.ComponentListener;
21 import java.util.HashMap;
22 import java.util.Map;
```

```
23
24 public class SimulatedLocationSensor extends SensorComponent
   implements ComponentListener {
25   private static long LEASE = 10000;
26   private Window window;
27   private MetadataBroadcastThread thread;
28   public SimulatedLocationSensor(Window window) {
29     super();      this.window = window;
30   }
31
32   public void initialise() throws CocoaException {
33     super.initialise();
34     Metadata metadata = new Metadata();
35     metadata.setLocation(new LocationSensorInformation(new
   SymbolicLocation(window.getName())));
36     metadata.setCoverage(new SymbolicCoverage(window.getName()));
37     metadata.setID(IDSensorInformation.generateID());
38     metadata.setType(new TypeSensorInformation("location"));
39     metadata.setLease(LEASE);
40     String[] names = {"value"};
41     int[] types = {FormatSensorInformation.STRING};
42     metadata.setFormat(new FormatSensorInformation(names,types));
43     this.setSensorMetadata(metadata);
44     window.addComponentListener(this);
45     thread = new MetadataBroadcastThread(this,5000);
46     thread.start();
47   }
48
49   public void finalise() throws CocoaException {
50     thread.stopProcess();
51     super.finalise();
52     window.removeComponentListener(this);
53   }
54
55   public void componentResized(ComponentEvent e) {}
56
57   public void componentMoved(ComponentEvent e) {
58     if(e.getSource() == window){
59       try {
60         RelativeLocation loc = new RelativeLocation(new
   SymbolicLocation("spacedog"),window.getLocation().x + (window.getWidth
   () / 2), window.getLocation().y + (window.getHeight() / 2));
61         Map values = new HashMap();
62         values.put("value",new StringSensorValue(loc.toString()));
63         Data data = new Data(values,LEASE,this.getSensorMetadata().
   getID());
64         super.update(data);
65       } catch (SensorException exp) {
66         exp.printStackTrace();
```

```
67          }
68        }
69    }
70
71    public void update() throws SensorException {}
72    public void componentShown(ComponentEvent e) {}
73    public void componentHidden(ComponentEvent e) {}
74  }
```

C.2 Activity Sensor for Simulated Environment

The source code in this section is of the sensor component used to represent the activity sensor in the simulated environment.

```
1
2  package ie.tcd.cs.dsg.karma.sensors;
3
4  import ie.tcd.cs.dsg.cocoa.cocoa.CocoaException;
5  import ie.tcd.cs.dsg.cocoa.contextmodel.SymbolicLocation;
6  import ie.tcd.cs.dsg.cocoa.sensor.FormatSensorInformation;
7
8  import ie.tcd.cs.dsg.cocoa.sensor.IDSensorInformation;
9  import ie.tcd.cs.dsg.cocoa.sensor.LocationSensorInformation;
10 import ie.tcd.cs.dsg.cocoa.sensor.Metadata;
11 import ie.tcd.cs.dsg.cocoa.sensor.SensorComponent;
12 import ie.tcd.cs.dsg.cocoa.sensor.Data;
13 import ie.tcd.cs.dsg.cocoa.sensor.SensorException;
14 import ie.tcd.cs.dsg.cocoa.sensor.StringSensorValue;
15 import ie.tcd.cs.dsg.cocoa.sensor.SymbolicCoverage;
16 import ie.tcd.cs.dsg.cocoa.sensor.TypeSensorInformation;
17 import ie.tcd.cs.dsg.cocoa.util.MetadataBroadcastThread;
18
19 import java.awt.Window;
20 import java.beans.PropertyChangeEvent;
21 import java.beans.PropertyChangeListener;
22 import java.util.HashMap;
23 import java.util.Map;
24 public class WindowActivitySensor extends SensorComponent implements
   PropertyChangeListener {
25   private static long LEASE = 10000;
26   private MetadataBroadcastThread thread;
27   private Window window;
28   public WindowActivitySensor(Window window) {
29     super();
30     this.window = window;
31   }
```

```
32
33   public void initialise() throws CocoaException {
34     super.initialise();
35     Metadata metadata = new Metadata();
36     metadata.setLocation(new LocationSensorInformation(new
  SymbolicLocation(window.getName())));
37     metadata.setCoverage(new SymbolicCoverage(window.getName()));
38     metadata.setID(IDSensorInformation.generateID());
39     metadata.setType(new TypeSensorInformation("active"));
40     metadata.setLease(LEASE);
41     String[] names = {"value"};
42     int[] types = {FormatSensorInformation.STRING};
43     metadata.setFormat(new FormatSensorInformation(names,types));
44     this.setSensorMetadata(metadata);
45     thread = new MetadataBroadcastThread(this,5000);
46     thread.start();
47    }
48
49   public void finalise() throws CocoaException {
50     thread.stopProcess();
51     super.finalise();
52   }
53   public void propertyChange(PropertyChangeEvent evt) {
54     try {
55       Map values = new HashMap();
56       values.put("value",new StringSensorValue(evt.getPropertyName()))
  ;
57       super.update(new Data(values,LEASE,this.getSensorMetadata().
  getID()));
58     } catch (SensorException e) {
59       e.printStackTrace();
60     }
61   }
62
63   public void update() throws SensorException {}
64 }
```

C.3 GPS Location Sensor

The source code in this section is of the sensor component used to represent the GPS location sensor.

```
1 package ie.tcd.cs.dsg.sensors; import java.io.BufferedReader;
2
3 import java.io.DataOutputStream;
4 import java.io.IOException;
```

```
5  import java.io.InputStreamReader;
6  import java.net.InetAddress;
7  import java.net.Socket;
8  import java.net.UnknownHostException;
9
10 import ie.tcd.cs.dsg.cocoa.cocoa.Cocoa;
11 import ie.tcd.cs.dsg.cocoa.cocoa.CocoaException;
12 import ie.tcd.cs.dsg.cocoa.cocoa.Environment;
13 import ie.tcd.cs.dsg.cocoa.contextmodel.SymbolicLocation;
14 import ie.tcd.cs.dsg.cocoa.sensor.Data;
15 import ie.tcd.cs.dsg.cocoa.sensor.DoubleSensorValue;
16 import ie.tcd.cs.dsg.cocoa.sensor.FormatSensorInformation;
17 import ie.tcd.cs.dsg.cocoa.sensor.IDSensorInformation;
18 import ie.tcd.cs.dsg.cocoa.sensor.LocationSensorInformation;
19 import ie.tcd.cs.dsg.cocoa.sensor.Metadata;
20 import ie.tcd.cs.dsg.cocoa.sensor.SensorComponent;
21 import ie.tcd.cs.dsg.cocoa.sensor.SensorException;
22 import ie.tcd.cs.dsg.cocoa.sensor.SymbolicCoverage;
23 import ie.tcd.cs.dsg.cocoa.sensor.TypeSensorInformation;
24 import ie.tcd.cs.dsg.cocoa.util.MetadataBroadcastThread;
25 import ie.tcd.cs.dsg.cocoa.util.SensorUpdateThread;
26
27 public class GPSPositionSensor extends SensorComponent {
28   private static final String LOGGER = "cocoa.logger";
29   private static final String PORT_NUMBER = "gpssensor.port";
30   private static final String SERVER = "gpsensor.server";
31   private static final String METADATA_LEASE = "gpsensor.metadata.
  lease";
32   private static final String DATA_LEASE = "gpsensor.data.lease";
33   private static final String SYMBOLIC_LOCATION = "gpsensor.
  symlocation";
34   private static final String PROTOCOL_POSITION = "p";
35   private static final String PROTOCOL_DATE = "d";
36   private static final String PROTOCOL_ALTITUDE = "a";
37   private static final String PROTOCOL_SPEED = "v";
38   private static final String PROTOCOL_STATUS = "s";
39   static final String PROTOCOL_MODE = "m";
40   private static final int DEFAULT_PORT_NUMER = 2947;
41   private static final String DEFAULT_SERVER = "127.0.0.1";
42   private static final long DEFAULT_METADATA_LEASE= 60000;
43   private static final long DEFAULT_DATA_LEASE = 10000;
44   private int portNumber;     private InetAddress server;
45   private Socket gpsSocket;
46   private BufferedReader input;
47   private DataOutputStream output;
48   private long metadataLease;
49   private long dataLease;
50   private boolean initialised = false;
51   private MetadataBroadcastThread metadataBroadcastThread;
```

```
52   private SensorUpdateThread updateThread;
53
54   //this sensor uses the gpsd daemon that can be found at http://www.
  pygps.org/gpsd/
55   public GPSPositionSensor() {
56     super();
57   }
58
59   public void initialise() throws CocoaException {
60     super.initialise();
61     Environment environment = (Environment) Cocoa.components.retrieve(
  Cocoa.ENVIRONMENT);
62     if(environment == null) {
63       throw new CocoaException("Environment component has not been
  loaded");
64     }
65     String dataLeaseString = environment.getProperty(DATA_LEASE);
66     if(dataLeaseString == null) {
67       this.metadataLease = DEFAULT_METADATA_LEASE;
68     }else {
69       try {
70         this.dataLease = Long.parseLong(dataLeaseString);
71       }catch(NumberFormatException e) {
72         throw new CocoaException("Data lease number for GPSensor has
  not been defined properly");
73       }
74     }
75     Metadata metadata = new Metadata();
76     String symbolLocation = environment.getProperty(SYMBOLIC_LOCATION)
  ;
77     if(symbolLocation == null) {
78       throw new CocoaException("Symbolic location for GPSSensor has
  not been defined.");
79     }
80     metadata.setLocation(new LocationSensorInformation(new
  SymbolicLocation(symbolLocation)));
81     metadata.setCoverage(new SymbolicCoverage(symbolLocation));
82     metadata.setID(IDSensorInformation.generateID());
83     metadata.setType(new TypeSensorInformation("location"));
84     String metadataLeaseString = environment.getProperty(
  METADATA_LEASE);
85     if(metadataLeaseString == null) {
86       this.metadataLease = DEFAULT_METADATA_LEASE;
87     }else {
88       try {
89         this.metadataLease = Long.parseLong(metadataLeaseString);
90       }catch(NumberFormatException e) {
91         throw new CocoaException("Metadata lease number for GPSensor
  has not been defined properly");
```

```
92        }
93      }
94      metadata.setLease(this.metadataLease);
95      String[] names = {"ns","ew"};
96      int[] types = {FormatSensorInformation.DOUBLE,
    FormatSensorInformation.DOUBLE};
97      metadata.setFormat(new FormatSensorInformation(names,types));
98      this.setSensorMetadata(metadata);
99      String portNumberString = environment.getProperty(PORT_NUMBER);
100     if(portNumberString != null) {
101       try {
102         this.portNumber = Integer.parseInt(portNumberString);
103       }catch(NumberFormatException e) {
104         throw new CocoaException("Port number for GPSensor has not
    been defined properly");
105       }
106     }else {
107       this.portNumber = DEFAULT_PORT_NUMER;
108     }
109     String serverString = environment.getProperty(SERVER);
110     try {
111       if(portNumberString == null) {
112         this.server = InetAddress.getLocalHost();
113       }else {
114         this.server = InetAddress.getByName(serverString);
115       }
116     } catch (UnknownHostException e) {
117       throw new CocoaException("Error determining server: " + e.
    toString());
118     }
119     try {
120       this.gpsSocket = new Socket(this.server, this.portNumber);
121       this.input = new BufferedReader(new InputStreamReader(this.
    gpsSocket.getInputStream()));
122       this.output = new DataOutputStream(this.gpsSocket.
    getOutputStream());
123     } catch (IOException e) {
124       throw new CocoaException("Error setting up connection with
    server: " + e.toString());
125     }
126     metadataBroadcastThread = new MetadataBroadcastThread(this,this.
    metadataLease/2);
127     metadataBroadcastThread.start();
128     updateThread = new SensorUpdateThread(this,this.dataLease / 2);
129     updateThread.start();
130     initialised = true;
131   }
132
133   public void finalise() throws CocoaException {
```

```
134     try {
135       if(this.initialised) {
136         super.finalise();
137         this.metadataBroadcastThread.stopProcess();
138         this.updateThread.stopProcess();
139         this.output.flush();
140         this.output.close();
141         output = null;
142         this.input.close();
143         input = null;
144         this.gpsSocket.close();
145         gpsSocket = null;
146       }
147     } catch (IOException e) {
148       throw new CocoaException("Error closing socket: " + e.toString()
    );
149     } finally {
150       this.initialised = false;
151     }
152   }
153
154   public void update() throws SensorException {
155     try {
156       if(this.initialised) {  //check if a valid poistion can be
    obtained
157         //get mode
158         this.output.writeBytes(PROTOCOL_MODE);
159         String modeString = this.input.readLine().trim();
160         if(!modeString.startsWith("GPSD,M")) {
161           throw new SensorException("Error retrieving gps data from
    server");
162       }
163       //Mode: 1=Fix not available  2=2D  3=3D
164       int mode = Integer.parseInt(modeString.substring(modeString.
    indexOf("=") +1));
165       if(mode == 1 || mode == 0) {
166         return;
167       }
168       //get status
169       this.output.writeBytes(PROTOCOL_STATUS);
170       String statusString = this.input.readLine().trim();
171       //GPS quality indicator (0=invalid; 1=GPS fix; 2=Diff. GPS fix)
172       if(!statusString.startsWith("GPSD,S")) {
173         throw new SensorException("Error retrieving gps data from
    server");
174       }
175       int status = Integer.parseInt(modeString.substring(modeString.
    indexOf("=") +1));
176       if(status == 0) {
```

```
177           return;
178           //once everything is ok then get gps position
179           this.output.writeBytes(PROTOCOL_POSITION);
180           String position = this.input.readLine().trim();
181           //data in the form GPSD,P=36.000000 123.000000\r\n
182           if(position.startsWith("GPSD,P")) {
183             int index = position.indexOf("=");
184             String [] parts = position.substring(index +1).split(" ");
185             DoubleSensorValue ns = new DoubleSensorValue(Double.
    parseDouble(parts[0]));
186             DoubleSensorValue ew = new DoubleSensorValue(Double.
    parseDouble(parts[1]));
187             Data data = new Data(this.dataLease,this.getSensorMetadata()
    .getID());
188             data.addSensorValue("ns",ns);
189              data.addSensorValue("ew",ew);
190             this.data = data;
191             this.update(data);
192           }
193         }
194       } catch (IOException e) {
195         e.printStackTrace();
196         throw new SensorException("Error retrieving gps data from server
    ");
197       } catch(NumberFormatException e) {
198         e.printStackTrace();
199         throw new SensorException("Error retrieving gps data from server
    ");
200       }
201     }
202 }
```

C.4 Punter Entity

Contained in this section is the script used for the punter entity presented in chapter 6 and the different configuration files for the entity used in the scenarios.

C.4.1 Script

```
1 punter extends person{
2   proximity(5)
3 }
```

C.4.2 Configuration File for Simulated Environment

```
1 //Components to load into framework
2 cocoa.components=ie.tcd.cs.dsg.comms.siena.SienaContextComms:ie.tcd.cs
  .dsg.comms.siena.SienaSensorComms:ie.tcd.cs.dsg.cocoa.util.
  SensorMetadataFilter:ie.tcd.cs.dsg.cocoa.util.SymbolicLocationFilter:
  ie.tcd.cs.dsg.cocoa.contextacquisition.ComponentContextAcquisition:ie.
  tcd.cs.dsg.jabs.compiler.Yabs:ie.tcd.cs.dsg.cocoa.runtime.
  ComponentStigmergyRuntime:ie.tcd.cs.dsg.karma.sensors.
  SimulatedLocationSensor
3
4 //Parameters to intialise context acquisition component
5 cocoa.model.drivers=ie.tcd.cs.dsg.PunterModelDriver
6 cocoa.model.name=puntermodel
7 cocoa.model.lease=5000
8
9 //Static context
10 cocoa.entity.person=peter
11
12 //Parameters to intialise scripting component.
13 cocoa.yabs.path=/opt/scripts
14 cocoa.yabs.script=punter
15
16 //Parameters to intialise Siena communication driver.
17 siena.master=tcp:spacedog.dsg.cs.tcd.ie:8461
18
19 //Parameters to intialise stigmergy runtime
20 cocoa.cv.ignoretime=true
21
22 /Other parameters for keeping track of symbolic information and
  logging
23 cocoa.staticsymboliclocations=/opt/runtimedata/staticsymboliclocations
24 cocoa.staticsymbolictimes=/opt/runtimedata/staticsymbolictimes
25 cocoa.logger=ie.tcd.cs.dsg.cocoa
```

C.4.3 Configuration File for the Jukebox Scenario

```
1 //Components to load into framework
2 cocoa.components=ie.tcd.cs.dsg.comms.siena.SienaContextComms:ie.tcd.cs
  .dsg.comms.siena.SienaSensorComms:ie.tcd.cs.dsg.cocoa.util.
  SensorMetadataFilter:ie.tcd.cs.dsg.cocoa.util.SymbolicLocationFilter:
  ie.tcd.cs.dsg.cocoa.contextacquisition.ComponentContextAcquisition:ie.
  tcd.cs.dsg.jabs.compiler.Yabs:ie.tcd.cs.dsg.cocoa.runtime.
  ComponentStigmergyRuntime:ie.tcd.cs.dsg.karma.sensors.
  SimulatedLocationSensor
3
4 //Parameters to intialise context acquisition component
5 cocoa.model.drivers=ie.tcd.cs.dsg.PunterModelDriver
6 cocoa.model.name=puntermodel   cocoa.model.lease=5000
7
8 //Static context
```

```
cocoa.entity.person=jane
cocoa.entity.music=hip-hop

//Parameters to intialise scripting component.
cocoa.yabs.path=/opt/scripts
cocoa.yabs.script=punter

//Parameters to intialise Siena communication driver.
siena.master=tcp:spacedog.dsg.cs.tcd.ie:8461

//Parameters to intialise stigmergy runtime
cocoa.cv.ignoretime=true

//Other parameters for keeping track of symbolic information and logging
cocoa.staticsymboliclocations=/opt/runtimedata/staticsymboliclocations
cocoa.staticsymbolictimes=/opt/runtimedata/staticsymbolictimes
cocoa.logger=ie.tcd.cs.dsg.cocoa
```

C.4.4 Configuration File for the Voice Scenario

```
//Components to load into framework
cocoa.components=ie.tcd.cs.dsg.comms.siena.SienaContextComms:ie.tcd.cs.dsg.comms.siena.SienaSensorComms:ie.tcd.cs.dsg.cocoa.util.SensorMetadataFilter:ie.tcd.cs.dsg.cocoa.util.SymbolicLocationFilter:ie.tcd.cs.dsg.cocoa.contextacquisition.ComponentContextAcquisition:ie.tcd.cs.dsg.jabs.compiler.Yabs:ie.tcd.cs.dsg.cocoa.runtime.ComponentStigmergyRuntime

//Parameters to intialise context acquisition component
cocoa.model.drivers=ie.tcd.cs.dsg.PunterModelDriver
cocoa.model.name=puntermodel
cocoa.model.lease=5000

//Static context
cocoa.entity.person=Vinny
cocoa.entity.location=F32, OReilly Instituation

//Parameters to intialise scripting component.
cocoa.yabs.path=/opt/scripts
cocoa.yabs.script=punter

//Parameters to intialise Siena communication driver.
siena.master=tcp:spacedog.dsg.cs.tcd.ie:8461

//Parameters to intialise stigmergy runtime
cocoa.cv.ignoretime=true

```

```
23 //Other parameters for keeping track of symbolic information and
   logging
24 cocoa.staticsymboliclocations=/opt/runtimedata/staticsymboliclocations
25 cocoa.staticsymbolictimes=/opt/runtimedata/staticsymbolictimes
26 cocoa.logger=ie.tcd.cs.dsg.cocoa
```

C.4.5 Configuration File for Westland Row

```
1  //Components to load into framework
2  cocoa.components=ie.tcd.cs.dsg.comms.steam.SteamContextComms:ie.tcd.cs
   .dsg.comms.steam.SteamSensorComms:ie.tcd.cs.dsg.cocoa.util.
   SensorMetadataFilter:ie.tcd.cs.dsg.cocoa.util.SymbolicLocationFilter:
   ie.tcd.cs.dsg.cocoa.contextacquisition.ComponentContextAcquisition:ie.
   tcd.cs.dsg.jabs.compiler.Jabs:ie.tcd.cs.dsg.cocoa.contextualview.
   ComponentCV:ie.tcd.cs.dsg.sensors.GPSPositionSensor
3
4  //Parameters to intialise context acquisition component
5  cocoa.model.drivers=ie.tcd.cs.dsg.PunterModelDriver
6  cocoa.model.name=puntermodel  cocoa.model.lease=5000
7
8  //Static context
9  cocoa.entity.person=Vinny
10
11 //Parameters to intialise scripting component.
12 cocoa.yabs.path=/opt/scripts  cocoa.yabs.script=punter
13
14 //Parameters to intialise Steam communication driver.
15 steam.mode=mobile
16 steam.range=50
17 steam.period=1
18 steam.proximity=50
19
20 //Parameters to intialise stigmergy runtime
21 cocoa.cv.ignoretime=true
22
23 //Other parameters for keeping track of symbolic information and
   logging
24 cocoa.staticsymboliclocations=/opt/runtimedata/staticsymboliclocations
25 cocoa.staticsymbolictimes=/opt/runtimedata/staticsymbolictimes
26 cocoa.logger=ie.tcd.cs.dsg.cocoa
27 gpsensor.symlocation=vinny
28 gpsensor.data.lease=10000
29 gpsensor.metadata.lease=60000
30 gpsensor.server=brain
31 gpssensor.port=2947
```

C.5 Help Assistant Entity

Contained in this section is a sample configuration file that can be used to for the help assistant entity presented in chapter 6.

C.5.1 Configuration File

```
1  //Components to load into framework
2  cocoa.components=ie.tcd.cs.dsg.comms.siena.SienaContextComms:ie.tcd.cs
   .dsg.comms.siena.SienaSensorComms:ie.tcd.cs.dsg.cocoa.util.
   SensorMetadataFilter:ie.tcd.cs.dsg.cocoa.util.SymbolicLocationFilter:
   ie.tcd.cs.dsg.cocoa.contextacquisition.ComponentContextAcquisition:ie.
   tcd.cs.dsg.jabs.compiler.Yabs:ie.tcd.cs.dsg.cocoa.runtime.
   ComponentStigmergyRuntime:ie.tcd.cs.dsg.karma.sensors.
   SimulatedLocationSensor
3
4  //Parameters to intialise context acquisition component
5  cocoa.model.drivers=ie.tcd.cs.dsg.AsisstantModelDriver
6  cocoa.model.name=assistantmodel
7  cocoa.model.lease=5000
8
9  //Static context
10 cocoa.entity.object=johnshelpassistant
11
12 //Parameters to intialise scripting component.
13 cocoa.yabs.path=/opt/scripts
14 cocoa.yabs.script=assistant
15
16 //Parameters to intialise Siena communication driver.
17 siena.master=tcp:spacedog.dsg.cs.tcd.ie:8461
18
19 //Parameters to intialise stigmergy runtime
20 cocoa.cv.ignoretime=true
21
22 //Other parameters for keeping track of symbolic information and
   logging
23 cocoa.staticsymboliclocations=/opt/runtimedata/staticsymboliclocations
24 cocoa.staticsymbolictimes=/opt/runtimedata/staticsymbolictimes
25 cocoa.logger=ie.tcd.cs.dsg.cocoa
```

C.6 Tourist Attraction Entity

Contained in this section is the script used for the tourist attraction entity presented in chapter 6 and the sample configuration file for the entity.

C.6.1 Script

```
touristspot extends place{
  proximity(5)
}
```

C.6.2 Configuration File

```
//Components to load into framework
cocoa.components=ie.tcd.cs.dsg.comms.siena.SienaContextComms:ie.tcd.cs
.dsg.comms.siena.SienaSensorComms:ie.tcd.cs.dsg.cocoa.util.
SensorMetadataFilter:ie.tcd.cs.dsg.cocoa.util.SymbolicLocationFilter:
ie.tcd.cs.dsg.cocoa.contextacquisition.ComponentContextAcquisition:ie.
tcd.cs.dsg.jabs.compiler.Yabs:ie.tcd.cs.dsg.cocoa.runtime.
ComponentStigmergyRuntime:ie.tcd.cs.dsg.karma.sensors.
SimulatedLocationSensor

//Parameters to intialise context acquisition component
cocoa.model.drivers=ie.tcd.cs.dsg.TouristSpotModelDriver
cocoa.model.name=touristspotmodel
cocoa.model.lease=5000

//Static context
cocoa.entity.place=trinitycollegedublin
cocoa.entity.about=http://www.tcd.ie

//Parameters to intialise scripting component.
cocoa.yabs.path=/opt/scripts
ocoa.yabs.script=touristspot

//Parameters to intialise Siena communication driver.
siena.master=tcp:spacedog.dsg.cs.tcd.ie:8461

//Parameters to intialise stigmergy runtime
cocoa.cv.ignoretime=true

//Other parameters for keeping track of symbolic information and
logging
cocoa.staticsymboliclocations=/opt/runtimedata/staticsymboliclocations
cocoa.staticsymbolictimes=/opt/runtimedata/staticsymbolictimes
cocoa.logger=ie.tcd.cs.dsg.cocoa
```

C.7 Electronic Tour Guide Entity

Contained in this section is a sample configuration file that can be used to for the electronic tour guide entity presented in chapter 6.

C.7.1 Configuration File

```
1 //Components to load into framework cocoa.components=ie.tcd.cs.dsg.
  comms.siena.SienaContextComms:ie.tcd.cs.dsg.comms.siena.
  SienaSensorComms:ie.tcd.cs.dsg.cocoa.util.SensorMetadataFilter:ie.tcd.
  cs.dsg.cocoa.util.SymbolicLocationFilter:ie.tcd.cs.dsg.cocoa.
  contextacquisition.ComponentContextAcquisition:ie.tcd.cs.dsg.jabs.
  compiler.Yabs:ie.tcd.cs.dsg.cocoa.runtime.ComponentStigmergyRuntime:ie
  .tcd.cs.dsg.karma.sensors.SimulatedLocationSensor
2
3 //Parameters to intialise context acquisition component
4 cocoa.model.drivers=ie.tcd.cs.dsg.TourGuideModelDriver
5 cocoa.model.name=tourguidemodel
6 cocoa.model.lease=5000
7
8 //Static context
9 cocoa.entity.object=janestourguide
10
11 //Parameters to intialise scripting component.
12 cocoa.yabs.path=/opt/scripts
13 cocoa.yabs.script=tourguide
14
15 //Parameters to intialise Siena communication driver.
16 siena.master=tcp:spacedog.dsg.cs.tcd.ie:8461
17
18 //Parameters to intialise stigmergy runtime
19 cocoa.cv.ignoretime=true
20
21 //Other parameters for keeping track of symbolic information and
  logging
22 cocoa.staticsymboliclocations=/opt/runtimedata/staticsymboliclocations
23 cocoa.staticsymbolictimes=/opt/runtimedata/staticsymbolictimes
24 cocoa.logger=ie.tcd.cs.dsg.cocoa
```

C.8 The Jukebox Entity

Contained in this section is a sample configuration file that can be used to for the jukebox entity presented in chapter 6.

C.8.1 Configuration File for Simulated Environment

```
1 //Components to load into framework
2 cocoa.components=ie.tcd.cs.dsg.comms.siena.SienaContextComms:ie.tcd.cs
  .dsg.comms.siena.SienaSensorComms:ie.tcd.cs.dsg.cocoa.util.
  SensorMetadataFilter:ie.tcd.cs.dsg.cocoa.util.SymbolicLocationFilter:
  ie.tcd.cs.dsg.cocoa.contextacquisition.ComponentContextAcquisition:ie.
```

```
tcd.cs.dsg.jabs.compiler.Yabs:ie.tcd.cs.dsg.cocoa.runtime.
ComponentStigmergyRuntime:ie.tcd.cs.dsg.karma.sensors.
SimulatedLocationSensor:ie.tcd.cs.dsg.karma.sensors.
SimulatedActivitySensor

//Parameters to intialise context acquisition component
cocoa.model.drivers=ie.tcd.cs.dsg.JukeboxModelDriver
cocoa.model.name=jukeboxmodel
cocoa.model.lease=5000

//Static context
cocoa.entity.object=thomasjukebox

//Parameters to intialise scripting component.
cocoa.yabs.path=/opt/scripts
cocoa.yabs.script=jukebox

//Parameters to intialise Siena communication driver.
siena.master=tcp:spacedog.dsg.cs.tcd.ie:8461

//Parameters to intialise stigmergy runtime
cocoa.cv.ignoretime=true

//Other parameters for keeping track of symbolic information and
logging
cocoa.staticsymboliclocations=/opt/runtimedata/staticsymboliclocations
cocoa.staticsymbolictimes=/opt/runtimedata/staticsymbolictimes
cocoa.logger=ie.tcd.cs.dsg.cocoa
jukebox.mp3dir=/opt/music/
```

C.8.2 Configuration File for Westland Row

```
//Components to load into framework cocoa.components=ie.tcd.cs.dsg.
comms.steam.SteamContextComms:ie.tcd.cs.dsg.comms.steam.
SteamSensorComms:ie.tcd.cs.dsg.cocoa.util.SensorMetadataFilter:ie.tcd.
cs.dsg.cocoa.util.SymbolicLocationFilter:ie.tcd.cs.dsg.cocoa.
contextacquisition.ComponentContextAcquisition:ie.tcd.cs.dsg.jabs.
compiler.Jabs:ie.tcd.cs.dsg.cocoa.contextualview.ComponentCV:ie.tcd.cs
.dsg.sensors.GPSPositionSensor

//Parameters to intialise context acquisition component
cocoa.model.drivers=ie.tcd.cs.dsg.JukeboxModelDriver
cocoa.model.name=jukeboxmodel
cocoa.model.lease=5000

//Static context
cocoa.entity.object=westcoastjukebox

//Parameters to intialise scripting component.
```

```
cocoa.yabs.path=/opt/scripts
cocoa.yabs.script=jukebox

//Parameters to intialise Steam communication driver.
steam.mode=mobile
steam.range=50
steam.period=1
steam.proximity=50

//Parameters to intialise stigmergy runtime
cocoa.cv.ignoretime=true

//Other parameters for keeping track of symbolic information and logging
cocoa.staticsymboliclocations=/opt/runtimedata/staticsymboliclocations
cocoa.staticsymbolictimes=/opt/runtimedata/staticsymbolictimes
cocoa.logger=ie.tcd.cs.dsg.cocoa
gpsensor.symlocation=westcoastjukebox
gpsensor.data.lease=10000
gpsensor.metadata.lease=60000
gpsensor.server=brain
gpssensor.port=2947
jukebox.mp3dir=/opt/music/
```

C.9 Streetlight Entity

Contained in this section is a sample configuration file that can be used to for the streetlight entity presented in chapter 6.

C.9.1 Configuration File

```
//Components to load into framework
cocoa.components=ie.tcd.cs.dsg.comms.siena.SienaContextComms:ie.tcd.cs.dsg.comms.siena.SienaSensorComms:ie.tcd.cs.dsg.cocoa.util.SensorMetadataFilter:ie.tcd.cs.dsg.cocoa.util.SymbolicLocationFilter:ie.tcd.cs.dsg.cocoa.contextacquisition.ComponentContextAcquisition:ie.tcd.cs.dsg.jabs.compiler.Yabs:ie.tcd.cs.dsg.cocoa.runtime.ComponentStigmergyRuntime:ie.tcd.cs.dsg.karma.sensors.SimulatedLocationSensor:ie.tcd.cs.dsg.karma.sensors.SimulatedActivitySensor

//Parameters to intialise context acquisition component
cocoa.model.drivers=ie.tcd.cs.dsg.StreetlightModelDriver
cocoa.model.name=streetlightmodel
cocoa.model.lease=5000

```

```
9  //Static context
10 cocoa.entity.object=streetlight1
11
12 //Parameters to intialise scripting component.
13 cocoa.yabs.path=/opt/scripts
14 cocoa.yabs.script=streetlight
15
16 //Parameters to intialise Siena communication driver.
17 siena.master=tcp:spacedog.dsg.cs.tcd.ie:8461
18
19 //Parameters to intialise stigmergy runtime
20 cocoa.cv.ignoretime=true
21
22 //Other parameters for keeping track of symbolic information and
   logging
23 cocoa.staticsymboliclocations=/opt/runtimedata/staticsymboliclocations
24 cocoa.staticsymbolictimes=/opt/runtimedata/staticsymbolictimes
25 cocoa.logger=ie.tcd.cs.dsg.cocoa
```

C.10 The Voice Entity

Contained in this section is a sample configuration file that can be used to for the voice entity presented in chapter 6.

C.10.1 Configuration File

```
1  //Components to load into framework
2  cocoa.components=ie.tcd.cs.dsg.comms.siena.SienaContextComms:ie.tcd.cs
   .dsg.comms.siena.SienaSensorComms:ie.tcd.cs.dsg.cocoa.util.
   SensorMetadataFilter:ie.tcd.cs.dsg.cocoa.util.SymbolicLocationFilter:
   ie.tcd.cs.dsg.cocoa.contextacquisition.ComponentContextAcquisition:ie.
   tcd.cs.dsg.jabs.compiler.Yabs:ie.tcd.cs.dsg.cocoa.runtime.
   ComponentStigmergyRuntime
3
4  //Parameters to intialise context acquisition component
5  cocoa.model.drivers=ie.tcd.cs.dsg.FirefoxModelDriver
6  cocoa.model.name=firefoxmodel
7  cocoa.model.lease=5000
8
9  //Static context
10 cocoa.entity.object=vinnysthevoice
11 cocoa.entity.location=F32, OReilly Instituation
12
13 //Parameters to intialise scripting component.
14 cocoa.yabs.path=/opt/scripts
15 cocoa.yabs.script=thevoice
```

```
16
17 //Parameters to intialise Siena communication driver.
18 siena.master=tcp:spacedog.dsg.cs.tcd.ie:8461
19
20 //Parameters to intialise stigmergy runtime
21 cocoa.cv.ignoretime=true
22
23 //Other parameters for keeping track of symbolic information and
   logging
24 cocoa.staticsymboliclocations=/opt/runtimedata/staticsymboliclocations
25 cocoa.staticsymbolictimes=/opt/runtimedata/staticsymbolictimes
26 cocoa.logger=ie.tcd.cs.dsg.cocoa
```

C.11 Siopa Entity

Contained in this section is a sample configuration file that can used for the siopa entity, presented in chapter 6, and the script to define its behavior in the environment.

C.11.1 Script

```
1 siopa extends place{
2   proximity(50)
3 }
```

C.11.2 Configuration File

```
1 //Components to load into framework
2 cocoa.components=ie.tcd.cs.dsg.comms.steam.SteamContextComms:ie.tcd.cs
  .dsg.comms.steam.SteamSensorComms:ie.tcd.cs.dsg.cocoa.util.
  SensorMetadataFilter:ie.tcd.cs.dsg.cocoa.util.SymbolicLocationFilter:
  ie.tcd.cs.dsg.cocoa.contextacquisition.ComponentContextAcquisition:ie.
  tcd.cs.dsg.jabs.compiler.Jabs:ie.tcd.cs.dsg.cocoa.contextualview.
  ComponentCV
3
4 //Parameters to intialise context acquisition component
5 cocoa.model.drivers=ie.tcd.cs.dsg.SiopaModelDriver
6 cocoa.model.name=siopamodel
7 cocoa.model.lease=5000
8
9 //Static context
10 cocoa.entity.place=spar
11 cocoa.entity.deals=http://172.16.8.254/spar.html
12 cocoa.entity.location=53°20'36\"N,6°4'59\"W
13
14 //Parameters to intialise scripting component.
15 cocoa.yabs.path=/opt/scripts
```

```
16 cocoa.yabs.script=punter
17
18 //Parameters to intialise Steam communication driver.
19 steam.location.lat=-614.9833
20 steam.location.lon=5320.6
21 steam.mode=fixed
22 steam.range=50
23 steam.period=1
24 steam.proximity=50
25
26 //Parameters to intialise stigmergy runtime
27 cocoa.cv.ignoretime=true
28
29 //Other parameters for keeping track of symbolic information and
   logging
30 cocoa.staticsymboliclocations=/opt/runtimedata/staticsymboliclocations
31 cocoa.staticsymbolictimes=/opt/runtimedata/staticsymbolictimes
32 cocoa.logger=ie.tcd.cs.dsg.cocoa
```

C.12 Shopping Assitant Entity

Contained in this section is a sample configuration file that can used for the shopping assitant entity presented chapter 6.

C.12.1 Configuration File

```
1 //Components to load into framework
2 cocoa.components=ie.tcd.cs.dsg.comms.steam.SteamContextComms:ie.tcd.cs
  .dsg.comms.steam.SteamSensorComms:ie.tcd.cs.dsg.cocoa.util.
  SensorMetadataFilter:ie.tcd.cs.dsg.cocoa.util.SymbolicLocationFilter:
  ie.tcd.cs.dsg.cocoa.contextacquisition.ComponentContextAcquisition:ie.
  tcd.cs.dsg.jabs.compiler.Jabs:ie.tcd.cs.dsg.cocoa.contextualview.
  ComponentCV:ie.tcd.cs.dsg.sensors.GPSPositionSensor
3 //Parameters to intialise context acquisition component
4 cocoa.model.drivers=ie.tcd.cs.dsg.ShoppingAssistantModelDriver
5 cocoa.model.name=shoppingassistantmodel
6 cocoa.model.lease=5000
7
8 //Static context
9 cocoa.entity.object=ciarashopptingassistant
10
11 //Parameters to intialise scripting component.
12 cocoa.yabs.path=/opt/scripts
13 cocoa.yabs.script=shoppingassistant
14
15 //Parameters to intialise Steam communication driver.
```

```
steam.mode=mobile
steam.range=50
steam.period=1
steam.proximity=50

//Parameters to intialise stigmergy runtime
cocoa.cv.ignoretime=true

//Other parameters for keeping track of symbolic information and logging
cocoa.staticsymboliclocations=/opt/runtimedata/staticsymboliclocations
cocoa.staticsymbolictimes=/opt/runtimedata/staticsymbolictimes
cocoa.logger=ie.tcd.cs.dsg.cocoa
gpsensor.symlocation=ciarashoppingassistant
gpsensor.data.lease=10000
gpsensor.metadata.lease=60000
gpsensor.server=brain
gpssensor.port=2947
```

Bibliography

[1] Gregory D. Abowd. Classroom 2000: An experiment with the instrumentation of a living educational environment. *IBM Systems Journal*, 38(4):508–530, October 1999.

[2] Mike Addlesee, Rupert Curwen, Joe Newman Steve Hodges, Pete Steggles, Andy Ward, and Andy Hopper. Implementing a sentient computing system. *IEEE Computer*, 34(8):50–56, August 2001.

[3] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *17th ACM SOSP*, Kiawah Island, SC, Dec 1999.

[4] J. F. Allen and G. Ferguson. Actions and events in interval temporal logic. *Journal of Logic and Computation*, 4(5):531–579, 1994.

[5] James F. Allen. Time and time again: the many ways to represent time. *International Journal of Intelligent Systems*, 6:341–355, 1991.

[6] J.F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, November 1983.

[7] Paramvir Bahl and Venkata N. Padmanabhan. Radar: An in-building RF-based user location and tracking system. In *IEEE INFOCOM*, March 2000.

[8] Peter Barron and Vinny Cahill. Using stigmergy to co-ordinate pervasive computing environments. In *Sixth IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'04)*, pages 62–71, December 2004.

[9] Peter Barron, Stefan Weber, Siobhan Clarke, and Vinny Cahill. Experiences deploying an ad-hoc network in an urban environment. In *IEEE ICPS Workshop on Multi-hop Ad hoc Networks: from theory to reality*, 2005.

[10] J. Barton and V. Vijayaraghavan. Ubiwise: A ubiquitous wireless infrastructure simulation environment. Technical Report HPL-2002-303, HP Labs, 2002. www.hpl.hp.com/techreports/2002/HPL-2002-303.pdf.

[11] R. Beckers, O.E. Holland, and J.L. Deneubourg. From location actions to global tasks: stigmergy and collective robotics. In *Artificial Life IV*, pages 181–189, 1994.

[12] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *Swarm Intelligence From Natural to Artificial Systems*. Oxford University Press, 1999.

[13] Eric Bonabeau, Andrej Sobkowski, Guy Theraulaz, and Jean-Louis Deneubourg. Adaptive task allocation inspired by a model of division of labor in social insects. In *Biocomputing and emergent computation: Proceedings of BCEC97*, pages 36–45. World Scientific Press, 1997.

[14] Eric Bonabeau and Guy Theraulaz. Swarm smarts. *Scientific American*, pages 72–79, March 2000.

[15] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, March 1986.

[16] Jason A. Brotherton. *Enriching Everyday Activities Through the Automated Capture and Access of Live Experiences - eClass: Building, Observing, and Understanding the Impact of Capture and Access in an Educational Domain*. PhD thesis, Georgia Institute of Technology, September 2001.

[17] P. J. Brown. The stick-e document: a framework for creating context-aware applications. In *EP'96*, 1996.

[18] P.J. Brown, J.D. Bovey, and X. Chen. Context-aware applications: From the laboratory to the marketplace. *IEEE Personal Communications*, 4(5):58–64, 1997.

[19] S. Brueckner. *Return from the Ant - Synthetic Ecosystems for Manufacturing Control*. PhD thesis, Humboldt University Berlin, 2000.

[20] Sven A. Brueckner and H. Van Dyke Parunak. Swarming agents for distributed pattern detection and classification. In *AAMAS*, 2002.

[21] B. Brumitt, B. Meyers, J. Krumm, A. Kern, and S. Shafer. Easyliving: Technologies for intelligent environments". In *Handheld and Ubiquitous Computing*, September 2000.

[22] B. Brumitt and S. Shafer. Better living through geometry. In *CHI Workshop on Situated Interaction in Ubiquitous Computing*, April 2000.

[23] B. Bullnheimer, R.F. Hartl, and C. Strauss. Applying the ant system to the vehicle routing problem. In *2nd Metaheuristics International Conference (MIC-97)*, Antipolis, France, 1997.

[24] V. Cahill, E. Gray, J.-M. Seigneur, C. Jensen, Y. Chen, B. Shand, N. Dimmock, A. Twigg, J. Bacon, C. English, W. Wagealla, S. Terzis, P. Nixon, G. Serugendo, C. Bryce, M. Carbone, K. Krukow, and M. Nielsen. Using Trust for Secure Collaboration in Uncertain Environments. In *Pervasive Computing Magazine*, volume 2, pages 52–61. IEEE, 2003.

[25] S. Camazine, J.L. Deneubourg, N. R. Franks, J. Sneyd, G. Theraulaz, and E. Bonabeau. *Self-organization in Biological Systems*. Princeton University Press, Princeton, NJ, 2001.

[26] R. Campbell and J. Krumm. Object recognition for an intelligent room. *IEEE Conf. on Computer Vision and Pattern Recognition*, june 2000.

[27] Gianni Di Caro and Marco Dorigo. Antnet: Distributed stigmergetic control for communications networks. *Artificial Intelligence Research*, 9:317–365, 1998.

[28] N. Carriero and D. Gelernter. Linda in context (parallel programming). *Comm. ACM*, 32(4):444–458, 1989.

[29] Antonio Carzaniga, David S. Rosenblum, and Alexander L Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, aug 2001.

[30] Paul Castro and Richard Muntz. Managing context data for smart spaces. *IEEE Personal Communications*, 7(5):44–46, October 2000.

[31] Renato Cerqueira, Carlos Cassino, and Roberto Ierusalimschy. Dynamic component gluing across different componentware systems. In *International Symposium on Distributed Objects and Applications (DOA'99)*, 1999.

[32] Harry Chen, Tim Finin, and Anupam Joshi. Using owl in a pervasive computing broker. In *Workshop on Ontologies in Agent Systems, AAMAS-2003*, July 2003.

[33] Keith Cheverst, Nigel Davies, Keith Mitchell, Adrian Friday, and Christos Efstratiou. Developing a context-aware electronic tourist guide: Some issues and experiences. In *CHI 2000*, pages 17–24, April 2000.

[34] M Coen. The future of human-computer interaction or how i learned to stop worrying and love my intelligent room. *IEEE Intelligent Systems*, March/April 1999.

[35] Michael Coen, Brenton Phillips, Nimrod Warshawsky, Luke Weisman, Stephen Peters, and Peter Finin. Meeting the computational needs of intelligent environments: The metaglue system. In *MANSE'99.*, Dublin, Ireland, 1999.

[36] Michael H. Coen. Building brains for rooms: Designing distributed software agents. In *IAAI'97*, pages 971–977, 1997.

[37] Open Geospatial Consortium. Sensor modeling language (sensorml) for in-situ and remote sensors. http://www.opengeospatial.org/specs/, Novemeber 2004.

[38] Antonio D'Angelo and Enrico Pagello. Using stigmerggy ti make emerging collective behaviors. In *8th Conference of the Italian Association for Artificial Intelligence*, Siena, Italy, September 2002.

[39] Nigel Davies, Keith Cheverst, Keith Mitchell, and Adrian Friday. Caches in the air: Disseminating information in the guide system. In *2nd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA '99)*, 1999.

[40] Nigel Davies and Hans-Werner Gellersen. Beyond prototypes: Challenges in deploying ubiquitous systems. *IEEE Pervasive Computing*, 1(1):26–35, January-March 2002.

[41] J.-L. Deneubourg, S. Aron, S. Goss, and J.-M Pasteels. The self-organizing exploratory pattern of argentine ant. *Journal of Insect Behavior*, 3:159–168, 1990.

[42] Anind Dey and Gregory Abowd. Towards a better understanding of context and context-awareness. In *Workshop on The What, Who, Where, When, and How of Context-Awareness, as part of the 2000 Conference on Human Factors in Computing Systems (CHI 2000)*, April 2000.

[43] Anind K. Dey and Gregory D. Abowd. The context toolkit: Aiding the development of context-aware applications. In *Workshop on Software Engineering for Wearable and Pervasive Computing (CHI '99)*, May 1999.

[44] Anind K. Dey, Daniel Salber, and Gregory D. Abowd. A context-based infrastructure for smart environments. In *1st International Workshop on Managing Interactions in Smart Environments (MANSE '99)*, 1999.

[45] M. Dorigo. *Optimization, Learning and Natural Algorithms*. PhD thesis, Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy, 1992.

[46] M. Dorigo, V. Maniezzo, and A. Colorni. The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics-Part B*, 26(1):29–41, 1996.

[47] Marco Dorigo, Gianni Di Caro, and Luca M. Gambardella. Ant algorithms for discrete optmization. *Artificial List*, 5(2):137–172, 1999.

[48] M. Doyle and M. Kalish. Indirect communication in multiple mobile autonomous agents.

In *IEEE Intelligent Systems: KSCO Special Edition (in press). Proc. International Conference on Knowledge Systems and Coalition Operations (KSCO-2004)*, Oct 2004.

[49] Maria R. Ebling, Guerney D. H. Hunt, and Hui Lei. Issues for context services for pervasive computing. In *Workshop on Middleware for Mobile Computing*, November 2001.

[50] W. Keith Edwards and Rebecca E. Grinter. At home with ubiquitous computing: Seven challenges. In *3rd international conference on Ubiquitous Computing*, pages 256–272, Atlanta, Georgia, USA, 2001. Springer-Verlag.

[51] T. Richardson et al. Virtual network computing. *IEEE Internet Computing*, 2(1):33–38, 1998.

[52] S. Fertig, E. Freeman, and D. Gelernter. Lifestreams: An alternative to the desktop metaphor. In *ACM SIGCHI Conference on Human Factors in Computing Systems Conference Companion (CHI '96)*, 1996.

[53] Margaret Fleck, Marcos Frid, Tim Kindberg, Eamonn O'Brien-Strain, Rakhi Rajani, and Mirjana Spasojevic. From informing to remembering: Ubiquitous systems in interactive museums. *IEEE Pervasive*, 1(2):13–21, 2002.

[54] Krzysztof Gajos. Rascal - a resource manager for multi agent systems in smart spaces. In *CEEMAS'01*, Kraków, Poland, 2001.

[55] Krzysztof Gajos and Ajay Kulkarni. FIRE: An information retrieval interface for intelligent environments. In *International Workshop on Information Presentation and Natural Multimodal Dialogue IPNMD 2001*, 2001.

[56] David Garlan, Dan Siewiorek, Asim Smailagic, and Peter Steenkiste. Project aura: Toward distraction-free pervasive computing. *IEEE Pervasive Computing*, 1(2), 2002.

[57] S. Goss, S. Aron, J. L. Deneubourg, and J. M Pasteels. Self-organized shortcuts in the argentine ant. *Naturwissenschaften*, 76:579–581, 1989.

[58] P.-P. Grassé. Le reconstruction du nid et les coordinations inter-individuelles chez bellicositermes natalensis et cubitermes sp. la theorie de la stigmergie: essai d'interpretation du comportement des termites constructeurs. *Insectes Sociaux*, 6:41–81, 1959.

[59] Cary G. Gray and David R. Cheriton. Lease: An efficient fault-tolerant mechanism for distributed file cache consistency. In *12th ACM Symposium on Operating System Principles*, pages 202–210, Litchfield Park, AZ, 1989.

[60] Robert Grimm. *System support for pervasive applications.* PhD thesis, University of Washington, 2002.

[61] Robert Grimm. One.world: Experiences with a pervasive computing architecture. *IEEE Pervasive Computing*, 3(3):22–30, July-September 2004.

[62] Robert Grimm, Janet Davis, Eric Lemar, Adam MacBeth, Steven Swanson, Thomas Anderson, Brian Bershad, Gaetano Borriello, Steven Gribble, and David Wetherall. System support for pervasive applications. *ACM Transactions of Computing Systems*, 22(4):421–486, November 2004.

[63] Laszlo Gulyas. Application of stigmergy - a coordination mechanism for mobile agents. In *1st Hungarian National Conference on Agent-Based Computing (HUNABC'98)*, pages 143–154, Budapest, 1999. Springer.

[64] Nicholas Hanssens, Ajay Kulkarni, Rattapoom Tuchinda, and Tyler Horton. Building agent-based intelligent workspaces. In *The International Workshop on Agents for Business Automation*, 2002.

[65] Andy Harter, Andy Hopper, Pete Steggles, Andy Ward, and Paul Webster. The anatomy of a context-aware application. In *5th Annual ACM/IEEE International Conference on Mobile Computing and Networking (Mobicom '99)*, 1999.

[66] Karen Henricksen, Jadwiga Indulska, and Andry Rakotonirainy. Modeling context information in pervasive computing systems. In *First International Conference on Pervasive Computing*, 2002.

[67] Christopher K. Hess. *The Design and Implementation of a Context-Aware File System For Ubquitous Computing Applications.* PhD thesis, University Of Illinois at Urbana-Champaign, 2003.

[68] Christopher K. Hess and Roy H. Campbell. A context-aware data management system for ubiquitous computing applications. In *International Conference of Distributed Computing Systems (ICDCS 2003)*, Providence, Rhode Island, May 19-22 2003.

[69] Christopher K. Hess, Manuel Roman, and Roy H. Campbell. Building applications for ubiquitous computing environments. In *International Conference on Pervasive Computing (Pervasive 2002)*, pages 16–29, August 2002.

[70] Owen Holland. Multiagent systems: Lessons from social insects and collective robotics. In *Coevolution and Learning in Multiagent Systems: Papers from the 1996 AAAI Spring Symposium*, pages 57–62, Menlo Park,CA, March 1996. AAAI Press. AAAI Technical Report SS-96-01.

[71] Owen Holland and Chris Melhuish. Stigmergy, self-organization, and sorting in collective robotics. *Artif. Life*, 5(2):173–202, 1999.

[72] L.E. Holmquist, F. Mattern, B. Schiele, P. Alahuhta, M. Beigl, and H.W. Gellersen. Smart-its friends: A technique for users to easily establish connections between smart artefacts. In *UBICOMP*, Sept 2001.

[73] IEEE. Wireless lan medium acsess control (mac) and physical layer (phy) specification (ieee 802.11). www.ieee.org, 1997.

[74] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. Lua-an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.

[75] Stephen S. Intille. Designing a home of the future. *IEEE Pervasive Computing*, 1(2), 2002.

[76] Stephen S. Intille and Kent Larson. Designing and evaluating supportive technology for

home. In *IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, 2003.

[77] Brad Johanson and Armando Fox. The event heap: A coordination infrastructure for interactive workspaces. In *Proc. 4th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA 2002)*, June 2002.

[78] Brad Johanson, Armando Fox, and Terry Winograd. The interactive workspaces project: Experiences with ubiquitous computing rooms. *IEEE Pervasive Computing Magazine*, 1(2), April-June 2002.

[79] Brad Johanson, Greg Hutchins, Terry Winograd, and Maureen Stone. Pointright: Experience with flexible input redirection in interactive workspaces. In *UIST '02*, 2002.

[80] G. Judd and P. Steenkiste. Providing contextual information to pervasive computing applications. In *IEEE International Conference on Pervasive Computing (PERCOM)*, Dallas, March 23-25 2003.

[81] Henricksen K, Indulska J, and Rakotonirainy. A infrastructure for pervasive computing: Challenges. In *Workshop on Pervasive computing INFORMATIK 01*, September 2001.

[82] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Next century challenges: Mobile networking for smart dust. In *Mobile Computing and Networking (MobiCom 99)*, August 1999.

[83] I. Kassabalidis, M.A. El-Sharkawi, R.J. Marks, P. Arabshahi, and A.A. Gray. Swarm intelligence for routing in communication networks. In *IEEE Global Telecommunications Conference*, 2001.

[84] James Kennedy and Russel C. Eberhart. *Swarm Intelligence*. Morgan Kaufmann, 2001.

[85] Cory D. Kidd, Robert J. Orr, Gregory D. Abowd, Christopher G. Atkeson, Irfan A. Essa, Blair MacIntyre, Elizabeth Mynatt, Thad E. Starner, and Wendy Newstetter. The aware home: A living laboratory for ubiquitous computing research. In *Second International Workshop on Cooperative Buildings - CoBuild'99*, 1999.

[86] Tim Kindberg and Armando Fox. System software for ubiquitous computing. *IEEE Pervasive Computing*, 1(1), 2002.

[87] Fabio Kon, Ashish Singhai, Roy H. Campbell, Dulcineia Carvalho, Robert Moore, and Francisco Ballesteros. 2k: A reflective, component-based operating system for rapidly changing environments. In *ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems*, 1998.

[88] G. Krasner and S. Pope. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of Object Oriented Programming*, 1(3):26–49, 1988.

[89] J. Krumm, S. Harris, B. Meyers, B. Brumitt, M. Hale, and S. Shafer. Multi-camera multi-person tracking for easyliving. In *IEEE Workshop on Visual Surveillance*, July 2000.

[90] Ajay Kulkarni. Design principles of a reactive behavioral system for the intelligent room. *Bitstream: The MIT Journal of EECS Student Research*, 2002.

[91] Sue Long, Rob Kooper, Gregory D. Abowd, and Christopher G. Atkeson. Rapid prototyping of mobile context-aware applications: The cyberguide case study. In *2nd ACM International Conference on Mobile Computing and Networking (MobiCom'96)*, November 1996.

[92] Christopher Lueg. On the gap between vision and feasibility. In *International Conference on Pervasive Computing (Pervasive 2002)*, Zurich, Switzerland., August 2002.

[93] M. Mamei and F. Zambonelli. Spreading pheromones in everyday environments via rfid technologies. In *2nd IEEE Symposium on Swarm Intelligence*, June 2005. to appear.

[94] Marco Mamei and Franco Zambonelli. Programming stigmergic coordination with the tota middleware. In *fourth international joint conference on Autonomous agents and multiagent systems*, pages 415 – 422, July 25 - 29 2005.

[95] V. Maniezzi, A. Colorni, and M. Dorigo. The ant system applied to the quadratic assignment problem. Technical Report IRIDIA/94-28, Universite Libre de Bruxelles, Belgium, 1994.

[96] Zachary Mason. Programming with stigmergy: Using swarms for construction. In *Artificial Life VIII: Proc of the Eighth International Conference on Artificial Life*, pages 371–374, 2002.

[97] Maja J. Mataric. Issues and approaches in the design of collective autonomous agents. *Robotics and Autonomous Systems*, 16:321–331, December 1995.

[98] Maja J. Mataric. Using communication to reduce locality in distributed multi-agent learning. *Journal of Experimental and Theorectical Artificial Intelligence*, 10(3):357–369, July-September 1998.

[99] R. Meier and V. Cahill. Steam: Event-based middleware for wireless ad hoc networks. In *International Workshop on Distributed Event-Based Systems (ICDCS/DEBS'02)*, pages 639–644, 2002.

[100] R. Meier and V. Cahill. Exploiting proximity in event-based middlewarefor collaborative mobile applications. In *4th IFIPInternational Conference on Distributed Applications and InteroperableSystems (DAIS'03), LNCS 2893*, pages 285–296, Paris, France, 2003. Springer-Verlag.

[101] Florian Michahelles, Erich Crameri, and Bernt Schiele. Design, Implementation and Testing of a Wearable Sensing System for Professional Downhill Skiing in Cooperation with Trainers. In *2nd International Forum on Applied Wearable Computing*, Zurich, Switzerland, March 2005.

[102] Florian Michahelles and Bernt Schiele. Sensing and monitoring professional skiing athletes: Lessons learned from a collaboration with ski trainer's. *IEEE Pervasive Computing Magazine*, July-September 2005.

[103] Sun Microsystems. Javacc. https://javacc.dev.java.net/.

[104] Sun Microsystems. Jini technology architectural overview. http://www.sun.com/jini/whitepapers/architecture.html.

[105] Ricardo Morla and Nigel Davies. Evaluating a location-based application: A hybrid test and simulation environment. *IEEE Pervasive Computing*, 3(3):48–56, July-September 2004.

[106] Ghita Kouadri Mostefaoui and Jacques Pasquier-Rocha. Context-aware computing: A guide for the pervasive computing community. In *IEEE/ACS International Conference on Pervasive Services (ICPS'04)*, 2004.

[107] M. Mozer. The neural network house: An environment that adapts to its inhabitants. In *AAAI Spring Symposium on Intelligent Environments*, pages 110–114, 1998.

[108] M. C. Mozer, R. H. Dodier, M. Anderson, L. Vidmar, R. F. Cruickshank III, and D. Miller. The neural network house: An overview. In L. Niklasson and M. Boden, editors, *Current trends in connectionism*, pages 371–380, 1995.

[109] M. C. Mozer and D. Miller. Parsing the stream of time: The value of event-based segmentation in a complex, real-world control problem. In *Adaptive processing of temporal information*, 1998.

[110] M. C. Mozer, L. Vidmar, and R. H. Dodier. The neurothermostat: Predictive optimal control of residential heating systems. In *Advances in Neural Information Processing Systems 9*, 1997.

[111] E. Mynatt, I. Essa, and W. Rogers. Increasing the oppurtunities for aging-in-place. In *ACM Conference on Universal Usability*, 2000.

[112] E.D. Mynatt, J. Rowan, S. Craighill, and A. Jacobs. Digital family portraits: Providing peace of mind for extended family members. In *ACM Conference on Human Factors in Computing Systems (CHI 2001)*, 2001.

[113] Alice Oh, Rattapoom Tuchinda, and Lin Wu. Meetingmanager: A collaborative tool in the intelligent room. In *Student Oxygen Workshop*, 2001.

[114] E. O'Neill, M. Klepal, D. Lewis, T. O'Donnell, D. O'Sullivan, and D. Pesch. A testbed for evaluating human interaction with ubiquitous computing environments. In *1st International Conference on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities (TRIDENCOM'05)*, pages 60–69, February 2005.

[115] J.K. Ousterhout. Scripting: higher level programming for the 21st century. *IEEE Computer*, 31(3):23–30, march 1998.

[116] H. Van Dyke Parunak. Making swarming happen. In *Swarming and Network Enable Command, Control, Communications, Computers, Intelligence, Surveillance and Reconnaissance (C4ISR)*, McLean, Virginia, USA, January 2003.

[117] H. Van Dyke Parunak, Sven A. Brueckner, Mitch Fleischer, and James Odell. Co-x: Defining what agents do together. In *Workshop on Teamwork and Coalition Formation, First Joint Conference on Autonmois Agents and Multiagent Systems (AAMAS'02)*, Bologna, Italy, August 2002.

[118] J. Pascoe. Adding generic contextual capabilities to wearable computers. *2nd International Symposium on Wearable Computers*, 1998.

[119] Charles E. Perkins and Elizabeth M. Royer. Ad hoc on-demand distance vector routing. In *2nd IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100, February 1999.

[120] Stephen Peters. Using intelligent spaces to capture meeting flow. In Submission, 2002.

[121] Brenton Phillips. *Metaglue: A Programming Language for Multi-Agent Systems*. PhD thesis, MIT, Cambridge, MA, 1999.

[122] C. Pinhanez and A. Bobick. Fast constraint propagation on specialized allen networks and its application to action recognition and control. MIT Tech Report 456, MIT, January 1998.

[123] C. Pinhanez and A. Bobick. It/i: A theater play featuring an autonomous computer

graphics character. In *ACM Multimedia'98 Workshop on Technologies for Interactive Movies*, January 1998.

[124] C. Pinhanez, K. Mase, and A. Bobick. Interval scritps: a design paradigm for story-based interactive systems. In *CHI'97*, March 1997.

[125] Claudio Santos Pinhanez. *Representation and Recognition of Action in Interactive Spaces*. PhD thesis, MIT, June 1999.

[126] Shankar R. Ponnekanti, Brad Johanson, Emre Kiciman, and Armando Fox. Portability, extensibility and robustness in iros. In *IEEE International Conference on Pervasive Computing and Communications (Percom 2003)*, March 2003.

[127] Shankar R. Ponnekanti, Brian Lee, Armando Fox, Pat Hanrahan, and Terry Winograd. Icrafter: A service framework for ubiquitous computing environments. In *Proc. Ubiquitous Computing Conference (UBICOMP)*, 2001.

[128] N.B Priyantha, A. Chakraborty, and H. Balakrishnan. The cricket location-support system. In *6th Ann. Int'l Conf. Mobile Computing and Networking (Mobicom 00)*, pages 32–43, 2000.

[129] Anand Ranganathan, Jalal Al-Muhtadi, Jacob Biehl, Brian Ziebart, Roy Campbell, and Brian Bailey. Towards a pervasive computing benchmark. In *PerWare '05 (Workshop on Middleware Support for Pervasive Computing) at the IEEE International Conference on Pervasive Computing and Communications (PerCom 2005)*, Kauai Island, Hawaii, March 8-12 2005.

[130] Anand Ranganathan and Roy H. Campbell. A middleware for context-aware agents in ubiquitous computing environments. In *ACM/IFIP/USENIX International Middleware Conference*, 2004.

[131] Manuel Roman and Roy Campbell. A middleware-based application framework for active space applications. In *ACM/IFIP/USENIX International Middleware Conference (Middleware 2003)*, Rio de Janeiro, Brazil, 2003.

[132] Manuel Roman and Roy H. Campbell. Gaia: Enabling active spaces. In *9th ACM SIGOPS European Workshop*, 2000.

[133] Manuel Roman, Christopher K. Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell, and Klara Nahrstedt. Gaia: A middleware infrastructure to enable active spaces. *IEEE Pervasive Computing*, 1(4):74–83, 2002.

[134] Manuel Roman, Christopher K. Hess, Anand Ranganathan Renato Cerqueira, Roy H. Campbell, and Klara Nahrstedt. Gaia: A middleware infrastructure to enable active spaces. *IEEE Pervasive Computing*, 1(4):74–83, Oct-Dec 2002.

[135] M. Satyanarayanan. Pervasive computing: vision and challenges. *IEEE Personal Communications*, 8(4):10 –17, Aug 2001.

[136] M. Satyanarayanan. The evolution of coda. *ACM Transactions on Computer Systems*, 20(2):85–124, May 2002.

[137] M. Satyanrayanan. Mobile information access. *IEEE Personal Comm*, 3(1):26–33, Feb 1996.

[138] B. Schilit, N. Adams, and R. Want. Context-aware computing applications. In *1st International Workshop on Mobile Computing Systems and Applications*, pages 85–90, 1994.

[139] Bill Schilit and M. Theimer. Disseminating active map information to mobile hosts. *IEEE Network*, 8(5):22–32, 1994.

[140] A. Schmidt. Implicit human computer interaction through context. *Personal Technologies*, 4(2), June 2000.

[141] Albrecht Schmidt. *Ubiquitous Computing - Computing in Context.* PhD thesis, Lancaster University, November 2002.

[142] Albrecht Schmidt, Kofi Asante Aidoo, Antti Takaluoma, Urpo Tuomela, Kristof Van Laerhoven, and Walter Van de Velde. Advanced interaction in context. In *1th Inter-*

national Symposium on Handheld and Ubiquitous Computing (HUC99), pages 89–101. Springer, 1999.

[143] Albrecht Schmidt, Martin Strohbach, Kristof van Laerhoven, and Hans-W. Gellersen. Ubiquitous interaction: Using surfaces in everyday environments as pointing devices. In *7th European Research Consortium for Informatics and Mathematics Workshop User Interfaces for All (UI4ALL 02)*, 2003.

[144] Jean Scholtz and Sunny Consolvo. Toward a framework for evaluating ubiquitous computing applications. *IEEE Pervasive Computing*, 3(2):82–88, April-June 2004.

[145] C. Schuckmann, L. Kirchner, J. Schummer, and J. Haake. Designing object-oriented synchronous groupware with coast. In *ACM 1996 Conference on Computer Supported Cooperative Work (CSCW'96)*, 1996.

[146] S. Shafer, J. Krumm, B. Brumitt, B. Meyers, M. Czerwinski, and D. Robbins. The new easyliving project at microsoft research. In *DARPA/NIST Workshop on Smart Spaces*, July 1998.

[147] Point Six. http://www.pointsix.com.

[148] Joao Pedro Sousa and David Garlan. Aura: an architectural framework for user mobility in ubiquitous computing environments. In *3rd Working IEEE/IFIP Conference on Software Architecture 2002*, August 2002.

[149] P. J. Steggles, P. M. Webster, and A. C. Harter. The implementation of a distributed framework to support 'follow me' applications. In *1998 International Conference on Parallel and Distributed Processing Technique and Applications (PDPTA'98)*, 1998.

[150] Peter Stone and Manuela M. Veloso. Multiagent systems: A survey from a machine learning perspective. *Autonomous Robots*, 8(3):345–383, 2000.

[151] Thomas Strang and Claudia Linnhoff-Popien. A context modelling survey. In *First International Workshop on Advanced Context Modelling, Reasoning And Management, UbiComp*, September 2004.

[152] Thomas Strang, Claudia Linnhoff-Popien, and Korbinian Frank. Cool: A context ontology language to enable contextual interoperability. In *4th IFIP International Conferece on Distributed Applications and Interoperable Systems (DAIS 2003)*, 2003.

[153] N.A. Streitz, J. Geißler, and T. Holmer. Roomware for cooperative buildings: Integrated design of architectural spaces and information spaces. In *CoBuild98*, February 1998.

[154] N.A. Streitz, J. Geißler, T. Holmer, S. Konomi, C. Müller-Tomfelde, W. Reischl, P. Rexroth, P. Seitz, and R. Steinmetz. i-land: An interactive landscape for creativitiy and innovation. In *ACM Conference on Human Factors in Computing Systems (CHI '99)*, 1999.

[155] Martin Strohbach, Hans-Werner Gellersen, Gerd Kortuem, and Christian Kray. Cooperative artefacts: Assessing real world situations with embedded technology. In *6th Int'l Conf. Ubiquitous Computing (Ubicomp 2004)*, pages 249–266, 2004.

[156] P. Tandler. Architecture of beach: The software infrastructure for roomware environments. In *CSCW 2000: Workshop on Shared Environments to Support Face-to-Face Collaboration*, December 2000.

[157] P. Tandler. Software infrastructure for ubiquitous computing environments: Supporting synchronous collaboration with heterogeneous devices. In *UbiComp 2001: Ubiquitous Computing, LNCS 2201*, Sep 30-Oct 2 2001.

[158] XMMS Team. Xmms multimedia system. http://www.xmms.org.

[159] G. Theraulaz and Bonabeau. Modelling the collective building of comples architectures in social insects with lattice swarms. *Journal of Theorecticak Biology*, 177:381–400, 1995.

[160] Guy Theraulaz and Eric Bonabeau. A brief history of stigmergy. *Artificial Life*, 5(2):97–116, 1999.

[161] Paul Valckenaers, Martin Kollingbaum, Hendrik Van Brussel, Olaf Bochmann, and

C. Zamfirescu. The design of muilt-agent coordination and control systems using stigmergy. In *IWES01*, March 2001.

[162] J. Waldo. The jini architecture for network-centric computing". *Comm. ACM*, 42(7):76–82, 1999.

[163] Xiao Hang Wang, Da Qing Zhang, and Hung Keng Pung. Ontology based context modeling and reasoning using owl. In *Second IEEE Annual Conference on Pervasive Computing and Communications Workshops*, 2004.

[164] R. Want, B. N. Schilit, N. I. Adams, R. Gold, K. Petersen, D. Goldberg, J. R. Ellis, and M. Weiser. An overview of the PARCTAB ubiquitous computing experiment. *IEEE Personal Communications*, 2(6):28–33, 1995.

[165] Andy Ward, Alan Jones, and Andy Hopper. A new location technique for the active office. *IEEE Personal Communications*, 4(5):42–47, October 1997.

[166] M. Weiser and J. Brown. The coming age of calm technology. *PowerGrid Journal*, 1.01, July 1996.

[167] Mark Weiser. The computer for the twenty-first century. *Scientific American*, pages 94–104, September 1991.

[168] Mark Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):74–84, 1993.

[169] Barry Brain Werger. Cooperation without deliberation: A minimal behavior-based approach to mult-robot teams. *Artificial Intelligence*, 110:293–320, 1998.

[170] Barry Brian Werger. The spirit of bolivia: Complex behavior through minimal control. In *RoboCup-97: The First ROBOT World Cup Soccer Games and Conferences*, 1997.

[171] Tony White and Bernard Pagurek. Towards multi-swarm problem sovling in networks. In *Third International Conference on Multi-Agent Systems (ICMAS'98)*, Paris, France, 1998.

[172] Brian C. Williams and P. Pandurang Nayak. Immobile robots: Artificial intelligence in the new millenium. *AI Magazine*, 17(3):16–35, 1996.

[173] Edward Osborne Wilson. *Sociobiology*. Belknap Press of Harvard University Press, 1975.

[174] P. Wyckoff, S. W. McLaughry, T. J. Lehman, and D. A. Ford. Tspaces. *IBM Systems Journal*, 37(3).

[175] Stephen S. Yau, Fariaz Karim, Yu Wang, Bin Wang, and Sandeep K.S. Gupta. Reconfigurable context-sensitive middleware for pervasive computing. *IEEE Pervasive Computing*, 1(3):33–40, 2002.