A Programming Model for Mobile, Context-Aware Applications

Gregory Biegel

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 2004

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.

Gregory Biegel

Dated: May 19, 2005

Permission to Lend and/or Copy

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

Gregory Biegel

Dated: May 19, 2005

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor, Prof. Vinny Cahill, for the invaluable expertise, interest, patience, and encouragement that he has invested in me over the past four years. Throughout my time in Dublin, Vinny has provided me with exceptional assistance and guidance in every respect, his faith in my ability never waning, even when my own did. I would also like to thank him for his expert arrangement of funding and adept management of the research group, allowing myself and others to concentrate on research.

To all of my colleagues in the Distributed Systems Group, I would like to say a huge thank you for the contributions made to this thesis, both direct and indirect, as well as for all the interesting debates, discussions, and fun over the years - being in DSG has been the highlight of my education, and about so much more than just distributed systems. I am especially grateful to Mélanie Bouroche for her patient and expert assistance with the sentient couch, to Neil O'Connor for his masterful help with the sentient car, and to Dr. René Meier for always making himself available to answer my questions on event systems. Thank you too to the rest of the F.32 lab, Dr. Raymond Cunningham, Barbara Hughes, Peter Barron, and Vinny Reynolds, for making it such a cool place to come to every morning. Thanks also go to all members of the CORTEX team within DSG for hours of fruitful discussions, as well as to the European project partner institutions I have had the privilege of working with and learning from.

To my parents half the world away in Zimbabwe, and my sister many thousands of miles away in Australia, thank you for the endless love, support, and optimism you have selflessly provided, despite facing your own challenges. Your strength is an inspiration.

The Beit Trust made it possible for me to come to Ireland in the first place through the generosity of a fellowship award, and I will always be grateful to the trustees for their faith in me, and for giving me the opportunity and rare privilege of attending one of the finest universities in Europe.

Last, but by no means least, thank you Hannah for the continuous love, patience, and understanding you have shown in the time it has taken me to complete this research.

Gregory Biegel

University of Dublin, Trinity College

October 2004

Abstract

Continuing advances in hardware miniaturisation and networking technologies have contributed to the widespread deployment of a range of mobile computing devices. Ad hoc communication between mobile devices enables the deployment of networks anywhere, without the need for pre-installed infrastructure. In contrast to the traditional stationary, desktop-bound model of computing, mobile computing applications experience frequent changes in location and execution environment, i.e., their context. At the same time, advances in miniaturization, and cost-effective fabrication of sensor technology, have led to the emergence of small, low-cost sensors, enabling applications to measure a host of environmental parameters. These advances have led to the emergence of a new class of application which may dynamically alter their behaviour based on the execution environment, as perceived through sensor input, in a computing paradigm known as *context-aware* computing. In addition to the complexity introduced by the dynamic nature of ad hoc networks, context-awareness raises a further set of challenges related to the acquisition and fusion of sensor data and the inference of appropriate behaviours.

Existing mobile context-aware applications tend be built in an application-specific manner and there is a lack of a generic, and commonly-accepted, programming model that may be applied across the domain. As a result, such applications remain difficult to build and deploy, with developers often having to deal with low-level issues not directly related to application development. Whilst a small number of approaches to supporting the development of such applications deal with the abstraction of low-level sensor data, there remains no generic programming model providing systematic support to the developer for the capture, representation, and use of context data.

This thesis describes a model for the development of context-aware applications in mobile ad hoc environments known as the *sentient object* model. A domain-specific programming model is developed, based on a set of high-level abstractions defined within the sentient object model, that provides a systematic approach to developing context-aware applications in mobile, ad hoc environments. The programming model supports the developer in the areas of sensor and actuator abstraction, sensor fusion and intelligent inference for applications operating in a mobile ad hoc communication environment.

In addition, this thesis describes a high-level graphical programming tool based on the sentient object model that exposes the programming model in an intuitive graphical manner, significantly reducing the need for the developer to write low-level syntax and aiding the rapid development of applications, through its code-generation ability.

The main contribution of this thesis is the provision of a domain-specific programming model that aids in the development of context-aware applications, in mobile ad hoc environments. The programming model assists in the realisation of a vision of pervasive computing by easing the development of mobile, context-aware applications and making application development accessible to a wider range of developers.

We validate our contribution through practical application of the programming model to representative context-aware applications. These applications illustrate how the sentient object model is applied in practice and validate the contribution of the model in easing the complexity of developing mobile, context-aware applications.

Publications Related to this Ph.D.

1. Aline Senart, Mélanie Bouroche, Gregory Biegel and Vinny Cahill *A Component-based Middleware Architecture for Sentient Computing* Workshop on Component-oriented approaches to Context-aware computing, ECOOP '04, Oslo, Norway, June 14 2004

2. Maomao Wu, Adrian Friday, Gordon Blair, Thirunavukkarasu Sivaharan, Paul Okandaj, Hector Duran-Limon, Carl-Fredrik Sørensen, Gregory Biegel and René Meier *Novel Component Middleware for Building Dependable Sentient Computing Applications* Workshop on Component-oriented approaches to Context-aware computing, ECOOP '04, Oslo, Norway, June 14 2004

3. Gregory Biegel and Vinny Cahill *A Framework for Developing Mobile, Context-aware Applications* in Proceedings of 2nd IEEE Conference on Pervasive Computing and Communications, Percom 2004, Orlando, FL, March 14-17 2004

4. Gregory Biegel and Vinny Cahill *Sentient Objects: Towards Middleware for Mobile, Context-aware Applications* European Research Consortium for Informatics and Mathematics, ERCIM News No. 54, July 2003

5. Vinny Cahill and Gregory Biegel *Sentient Objects for Context-aware Business Process Management* 2003 SAP Innovation Congress, Miami, Florida February 2003

6. Adrian Fitzpatrick, Gregory Biegel, Siobhán Clarke and Vinny Cahill *Towards a Sentient Object Model* Position Paper Workshop on Engineering Context-Aware Object Oriented Systems and Environments (ECOOSE), Seattle WA, USA November 2002

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

There remains no generic and commonly-accepted programming model supporting the requirements for the development of context-aware computing applications in mobile, ad hoc environments, resulting in the continuing adoption of impromptu, application-specific approaches by application developers.

This thesis describes a programming model that is specifically designed to support and ease the development of context-aware applications in mobile, ad hoc environments, making the development of such applications accessible to a wider audience, and in the process aiding in the realisation of the vision of pervasive computing [Wei91].

This introductory chapter explores the origins of pervasive computing and the importance of context-awareness as a subset of pervasive computing. The aims and objectives of the thesis are outlined, and our contribution defined.

1.1 Pervasive computing

There have been a number of clearly discernible eras in the field of computing since the development of modern electronic computers in the 1940s. The first era was characterised by large mainframe-based computing, leading up to the minicomputers of the 1960s and early 1970s. The development of the microprocessor in 1971 began what can only be described as a revolution in the industry characterised by the emergence of the personal computer in the

1980s. As Saffo noted [Saf97], up until the 1990s the industry was defined by *processing* of data by independent machines. The next discernible era was one of *access* to information via the global Internet, precipitated by cheap lasers delivering large quantities of bandwidth [Saf97] and the development of the Hypertext Markup Language (HTML) [BLC95] and the Hypertext Transfer Protocol (HTTP) [BLFF96], enabling global deployment of the World-Wide Web, as the favourite application of the Internet. The new millennium has witnessed the emergence of mobile computing symbolized by compact, portable devices networked via wireless communication interfaces. Mobile computing is still primarily concerned with access to information, addressing inherent challenges such as security and resource-poverty.

A new era is now emerging, both driven by and dependent on mobile computing. This era is variously termed **pervasive** [Sat01, NAU93], **ubiquitous** [Wei91], or **invisible** [Bor00] computing and may be defined as the saturation of the physical environment with computing and communication ability and the graceful integration of these systems with human users [Sat01]. As such, pervasive computing is predicated on the universal availability of vast numbers of cheap and compact devices, both mobile and stationary, dispersed throughout the physical environment. A variety of very similar interpretations of pervasive computing have been made, but it is commonly accepted within the research community that the field lies at the intersection of hardware miniaturisation and networking. A number of research challenges exist in the broad field of pervasive computing, but this thesis is primarily concerned with the *interaction* of computers with their physical environment. Interaction in this sense is defined as awareness of the physical environment via sensors, combined with actuation on the environment via actuators, in a context-aware manner. We envisage such interaction taking place opportunistically via mobile ad hoc networks operating in a dynamic environment.

1.2 Pervasive computing as environmental interaction

There are a number of factors that contribute to the perceived importance of interaction with the environment in the emerging field of pervasive computing. First and foremost, saturation of the environment with computing devices make environmental parameters an important system input and enables applications to be aware of the context in which they

operate. The importance of environmental awareness is truly realised when we consider mobility and consequently a rapidly changing environment. Whereas stationary desktop computers of the last era operated in a constant physical environment, rarely if ever changing location, administrative domain, or proximity to other devices, the opposite is true of mobile devices. Ad hoc mobility causes rapid changes in execution environment and awareness of these changes can be used to enhance the flexibility, adaptiveness, effectiveness, and efficiency of existing applications, whilst making a host of new applications possible, e.g., [SSF+03]. Furthermore, although the value of environmental awareness is realised in mobile computing, the mechanisms by which it is achieved are due to recent progress in the miniaturisation and fabrication of sensor components. Advances in manufacturing processes leading to low cost-to-performance ratios coupled with novel signal processing methods and high-speed, low-cost electronic circuits have provided cheap, compact sensors able to measure a range of environmental parameters [TK01].

In addition to sensing of the environment, devices may also be provided with a means to effect environmental changes via electro-mechanical actuators, enabling true environmental interaction. Associated advances in the fabrication of components including piezo materials and Micro-Electro-Mechanical Systems (MEMS), have led to the growing emergence of cheap and compact actuators.

1.2.1 Context

A central tenet of pervasive computing is that the computing infrastructure should fade into the background of consciousness and become part of the environment [Wei91]. Current infrastructure clearly falls well short of this goal with a great deal of explicit input still required from a small set of devices (e.g., mouse and keyboard). For infrastructure to truly fade into the background, application components which act autonomously and proactively based solely on the acquisition of information from the environment and their own knowledge, are necessary. *Context* is the commonly accepted term used to describe the state of the environment in which an application operates and in order to be minimally intrusive, an application needs to be context-aware, defined as the ability to sense and react to context. Context is potentially

made up of a number of attributes describing location, identity, activity, bandwidth, power and a host of other parameters. Extensive treatment of the definition of both context and context-awareness are offered in section 2.2.1.

Existing context-aware applications typically rely on a set of sensors that deliver information gathered from the environment, to the application. Many applications, e.g., Active Badge [HH94], FieldNote [RJD99], and GUIDE [DMCF99] rely exclusively on location sensors to influence application behaviour, whilst other applications such as TRIP [dIn99], and Smart-Its [GSB02], use multiple, heterogeneous sensors to measure the environment. Such applications are challenging to develop, requiring developers to interact with low-level sensor protocols, and devise schemes to fuse sensor data, in addition to writing application logic.

Several approaches have been proposed to ease the development of context-aware applications, predominantly through abstracting away from the complexity of low-level heterogeneous sensors. Most notable amongst these is the Context Toolkit [SDA99], which separates context acquisition from the rest of the application, through abstractions known as context widgets. Other approaches, such as Gaia [RHC+02], provide support for sensor fusion and intelligent inference in addition to sensor abstraction. A significant disadvantage of existing approaches to supporting context-aware applications is that although interaction with sensors is simplified, the application developer still has to know the source of sensor data at development time. In mobile ad hoc networks, this assumption that the identities and addresses of various data sources is known in advance does not hold, and consequently such applications require a highly decoupled communication method [PRJ04].

In addition, current approaches to supporting the development of context-aware applications have a disproportionate focus on environmental sensing, neglecting actuation on the environment and rather focusing exclusively on the manipulation of user interfaces. The realisation of technology which truly *fades* into the background indicates a more autonomous style of interaction is required, with less emphasis on user interaction.

Despite the maturity and availability of enabling technologies, the development of pervasive computing applications remains a highly application-specific process, with a lack of systems and services support and no generally-accepted programming model available. This

thesis recognises existing limitations and proposes a programming model providing systematic support for the development of context-aware applications in mobile, ad hoc environments.

1.3 Aims and objectives

The broad aim of this thesis is to provide a generic programming model to support the development of context-aware pervasive applications in mobile, ad hoc environments. We propose that awareness of the environment via sensors coupled with environmental interaction via actuators together constitute context-awareness as the basis of pervasive computing. Furthermore, we assert that pervasive computing is both predicated on, and dependent upon, mobile, ad hoc networks of devices. Static and infrastructure-based networks neither have the flexibility to support dynamic interactions between devices, nor provide the mobility patterns that make the physical environment an important input to the system.

In order to achieve our aim, we have identified a number of characteristics of applications that should be supported by our model:

- Sentience. The model should support development of applications with the ability to perceive the state of the surrounding environment through diverse, multi-modal sensors.
- Decentralisation. The model should support inherently distributed applications with no centralised processing or control.
- Extensibility. The model should be extensible, implying loose coupling between components and an ability to easily incorporate additional functionality.
- Scalability. The ability for applications to scale to the very large and dynamic numbers of devices envisaged in pervasive environments should be provided by the model.
- Usability/applicability. Pervasiveness implies popular adoption and consequently implementations of the model should be powerful enough to enable sophisticated application and yet remain usable to ensure accessibility to a wide audience.

1.4 The sentient object model

In this thesis, we describe a generic object model for the development of context-aware applications in mobile, ad hoc environments. This model is known as the *sentient object model* and is based on sentient objects - mobile intelligent software components that accept input from a variety of different sensors and interact with the environment via a variety of actuators. The sentient object model was proposed within the CORTEX[1] project, which also examined other aspects of pervasive computing in real-time environments.

The sentient object model defines software abstractions that ease the use of sensor information, and associated actuation by context-aware applications. The model provides a systematic approach to the challenges of context capture, fusion, representation, intelligent inference, and actuation.

The challenges introduced by ad hoc mobile computing environments in particular, are addressed by the use of an event-based communication mechanism, which does not rely on centralised control and provides loose coupling between objects, supporting object mobility and application evolution.

1.5 Contribution

Within this thesis we have developed a programming model that supports the development of context-aware pervasive applications in mobile, ad hoc environments. Our programming model develops the sentient object model, as an architectural model for mobile, context-aware applications, and provides a graphical application development tool, easing the development of applications based on the sentient object model. Our programming model supports the application developer in the following key ways:

- It provides abstractions for sensors and actuators, thus relieving the developer of the burden of low level interaction with various hardware devices.

[1]The CORTEX (CO-operating Real-time senTient objects: architecture and EXperimental evaluation) project is supported by the Future and Emerging Technologies programme of the Commission of the European Union under research contract IST-FET-2000-26031, http://cortex.di.fc.ul.pt

- It provides a probabilistic mechanism for fusing multi-modal fragments of sensor data together in order to derive higher-level context information.
- It provides an efficient approach to intelligent reasoning based on a hierarchy of contexts.
- It provides an event-based communication mechanism, designed for mobile ad hoc networks, for interaction between sensors, objects and actuators.
- It provides an easily accessible visual programming tool for developing applications reducing the need to write code.

The programming model fulfills the two major goals recently identified by Dey and Sohn [DS03] as being necessary for the successful development of ubiquitous, context-aware applications, namely:

1. Applications are easier to design, prototype and test, supporting a faster iterative development process.
2. Designers and end-users are empowered to build their own applications.

The main contribution of the thesis is the provision of generic support for the development of context-aware applications in mobile ad hoc environments. Our programming model provides a systematic approach to application design and implementation, and in doing so, aids the realisation of the vision of truly pervasive computing by significantly easing application development, and making it accessible to a wider audience.

1.6 Roadmap

The remainder of this thesis is organised as follows. Chapter 2 gives an overview of the area in which the research is performed, providing necessary background and introduces a set of requirements for supporting development of mobile, context-aware applications. Chapter 3 describes state-of-the-art approaches to supporting the development of such applications, analysing the extent to which they implement the requirements identified in Chapter 2.

Chapter 4 introduces our architectural model, the sentient object model, and describes it in detail. Chapter 5 describes the implementation of a graphical programming tool based on the sentient object model. In Chapter 6 we evaluate our model by applying it to the development of representative applications. Finally, Chapter 7 concludes and details open questions and opportunities for further research.

Chapter 2

Overview of the Research Area

This chapter provides an overview of the research area with which this thesis is concerned. The intention is to provide a background for the discussion of related work in Chapter 3, and the programming model presented in Chapter 4.

Mobile computing is introduced as an important technology underlying context-aware computing due to the fact that mobility causes frequent and interesting changes in application context, which may be used to proactively influence application behaviour. The challenges posed by mobile computing in general are characterised with respect to application development, and the importance of mobile, ad hoc networking in the realisation of pervasive computing is outlined, as well as the challenges imposed by this form of communication.

An extensive treatment of *context* and *context-awareness*, as well as discussion of the important issues arising in the development of context-aware applications follows. Based on this discussion, a set of requirements for the provision of generic support for the development of context-aware applications for mobile ad hoc environments, is derived.

2.1 Mobile computing

The remarkable advances made in the last decade in the fields of portable computers and wireless communications precipitated what is generally accepted as a major advance in information technology, namely, the emergence of *mobile computing*. The continuing relentless

advance towards ever smaller and more powerful devices enabled by faster microprocessors, more memory and greater storage at a greater economy is matched by an increase in the bandwidth and global coverage of wireless networks, leading to increased interconnection between devices. We define mobile computing as networked computing that uses common carrier frequencies to permit wireless devices to move within the broadcast coverage area whilst remaining connected to the network. Additionally, we constrain our definition of mobile computing to *on-line* mobile computing, that is computing dependent on a real-time, live, network connection[1], with short periods of disconnected operation.

Although truly pervasive computing could conceivably be realised to some degree through widespread deployment of fixed computing and networking technology, we take the view that this is highly unlikely, due to two major considerations. Firstly, the cost of deploying fixed ubiquitous networking infrastructure throughout the environment is prohibitive, and secondly it would be physically impossible to network mobile entities such as vehicles and aircraft in this way. Current trends indicate the increasingly widespread adoption of wireless networking between mobile devices, and we believe future advances in pervasive computing will be based predominantly on mobile computing. Furthermore, we believe that these advances will be based on *ad hoc* mobile networks, obviating the need for extensive deployment of gateway infrastructure.

Additionally, in a consideration biased towards context-aware computing as a subset of pervasive computing, mobility causes frequent changes to the context in which an application executes. In marked contrast to stationary systems, mobile systems may experience rapid changes in location, administrative domain, bandwidth availability and economy, temperature, speed, proximity to other devices, and a host of other environmental parameters. Related to this consideration is the fact that awareness of the dynamic execution context by an application on a mobile device allows the application to initiate specific activity, for instance, reallocation of resources. As a result, mobile computing environments exhibit a range of characteristics that both challenge the developer of applications for such environments, as well as providing a source of input to applications that may be used to control behaviour.

[1] In contrast to off-line processing on mobile devices coupled with intermittent synchronisation of data at points of connectivity.

	Network type	Nominal bandwidth
Wired	Gigabit Ethernet	1000 Mbps
	Fast Ethernet	100 Mbps
	DSL	1.5 Mbps
Wireless	IEEE 802.11b (2.4 GHz)	11 Mbps
	Bluetooth	to 1 Mbps †
	GPRS	0.056 Mbps ‡

Table 2.1: Comparison of bandwidth in wired and wireless networks
†Raw data rate
‡Nominal bandwidth at mobile speeds

2.1.1 Characteristics of mobile computing

Mobile computing poses a set of fundamental technical challenges to software design stemming primarily from the use of wireless communication (limited bandwidth), the ability to change locations (address management), and the need for portability of the device (resource constraints). There has been extensive research carried out in the field of mobile computing and the challenges posed therein have been understood for some time [IB92, Duc92, FZ94, Sat96]. As Badrinath et al. note, although these constraints are becoming less noticeable, the portability of mobile devices will always induce additional constraints relative to stationary computing [BW95].

Bandwidth and latency

The wireless networks on which mobile computing is predicated invariably provide less bandwidth than wired networks. By way of comparison, widely deployed fast Ethernet (100 Mbps) provides bandwidth almost ten times that of consumer grade wireless networking technology (11 Mbps). Table 2.1 illustrates the difference in bandwidth available in wired and wireless networks by listing the nominal bandwidth of a subset of wired and wireless network technologies. In practice, the bandwidth available on wireless channels is often less than the nominal rate due to interference on the channel.

Limited bandwidths may be effectively further reduced by high *latency*, defined as the

temporal delay between the transmission of data and its reception. Wireless networks typically have a high degree of latency, due to factors such as radio interference, and the fact that network protocols are often optimised for wired networks. For instance, the Transmission Control Protocol (TCP) [(Ed81b] performs poorly in wireless networks [BSAK95] due to assumptions made regarding packet loss. Most TCP implementations interpret packet loss as being a result of network congestion, rather than lost packets, and consequently reduce transmission rates to reduce the network load[2]. In the case of wireless networks, packet loss is generally due to the unreliability of the wireless link and a better response to packet loss would be to increase packet transmission rate in order to increase overall throughput [Tan96]. TCP throughput in mobile, ad hoc networks (see section 2.1.2) was found to drop significantly when node mobility caused link failures, due to TCP's inability to differentiate between link failure and congestion [HV02].

Another characteristic of mobile devices is that they often have multiple network interfaces and the bandwidth available to them is highly variable dependant on which interface is active. For instance, the HP iPAQ H5550, a popular consumer Personal Digital Assistant (PDA), has an infrared interface, Bluetooth, and 802.11b WiFi, all of which offer different bandwidth and latency at different ranges and different economy.

Consideration of the limited, variable bandwidth and high latency characteristics of wireless networks are important to the developer of mobile, context-aware applications for two major reasons. Firstly, awareness of the constraints of the underlying communication mechanisms by the developer allow appropriate design decisions to be taken so that best possible use is made of the limited resources. Secondly, awareness of the current state of the communication system by the application allows application-specific activity to be performed at run-time in a context-aware manner. In other words, the state of the communication substrate contributes to the context of the application.

[2]Due to Jacobson's slow start algorithm [Jac88]

Resource poverty

Despite recent and continuing advances in the design and fabrication of components, mobile devices will continue to have less resources available to them than stationary computers. Some resources, such as a smaller user interface and less storage capacity, are due to the very nature of the mobile device, but probably the biggest constraint to resource availability in mobile devices remains power. The fundamental trade-off is between batteries that are light enough to be truly mobile, yet still have power enough to enable useful computation. Both Badrinath et al. [BAI93] and Satyanarayanan [Sat96] observe the challenges of powering a mobile device, whilst Forman et al. [FZ94] note that the power consumption of dynamic components is proportional to CV^2F where C is the capacitance of the wires, V is the voltage, and F is the clock frequency. Power consumption can thus be reduced through a reduction in any of C, V or F. Up until recently, microchip manufacturers have focused on increasing F for stationary systems, but are now concentrating on reducing V in view of power constraints on mobile systems. The Pentium® M processor incorporated in Intel's Centrino^{TM} architecture significantly reduces the voltage of the chip and provides accompanying power savings [Cor03].

The resource poverty inherent in mobile computing is important to the developer of mobile, context-aware applications for much the same reasons as awareness of limited bandwidth and high latencies. Awareness of constrained resources both allows developers to design around the fact, as well as providing an additional source of context data to the application at run-time. Whereas system power might not be considered as an interesting input to a stationary system, it becomes a highly valuable input in a mobile system, allowing for example an application to minimise communication to conserve failing batteries.

Address and locality migration

Mobile devices change their point of connection to the network infrastructure frequently, in contrast to stationary systems which seldom, if ever, make such changes. The hierarchical routing system in use by the Internet Protocol (IP) [(Ed81a] uses host addresses which belong to specific physical networks, and as such the physical location of the host is encoded in its address.

The Domain Name System (DNS) [Moc87] provides some level of indirection through mapping the network address of a host to a name [Haa03], but addresses are typically cached with a long expiration time and with no way to invalidate out-of-date entries, with the result that human intervention is required to coordinate the use of addresses [FZ94]. This is highly undesirable in a ubiquitous computing environment where the intention is to have computation fade into the background.

Much effort has been made towards providing support for mobility, at both the *macro mobility* (mobility between administrative domains) and *micro mobility* (mobility within an administrative domain) levels, at various positions within the protocol stack [RB01]. A popular approach to supporting macro mobility at the network layer, Mobile IP [Per97], deals with addressing problems through the four basic mechanisms of broadcast, centralised location registers, home bases and forwarding pointers [FZ94]. Whilst Mobile IP provides a solution to macro mobility issues, it has the disadvantages of introducing extra latency and a centralised location register.

The issues raised by address and locality migration in mobile environments are important to developers of mobile, context-aware applications both in terms of the potential challenges in addressing application components, and in terms of the additional context information that such migration may provide to an applications.

Security

Security in mobile computing environments is important for three major reasons. Firstly, the air interface (the physical layer) of wireless networks is more difficult to secure and more prone to eavesdropping, which may compromise the privacy of data communications. The recent proliferation of wireless LANs has created an entire subculture dedicated to mapping the coverage and security characteristics of these networks[3]. Such activities are not confined to the benign activity of mapping wireless coverage, but may extend to malicious attacks on network resources. The shared wireless medium is also more vulnerable to denial of service (DoS) attacks which are more difficult to control than in wired networks. Such DoS attacks

[3] http://www.wardriving.com

may not just be aimed at denying access to the communication medium, but additionally at depleting the sparse power resources of nodes under attack. Lastly, the very nature of mobile devices makes them more prone to theft and accidental damage.

Security considerations are important to the developers of mobile, context-aware applications that will need to interact with unknown entities, often whilst disconnected from their home networks and associated security infrastructures. Applications will have to be able to take autonomous security decisions without reliance on security infrastructures such as certificate authorities and authorization servers [CGS+03]. One approach being explored in providing decentralised security management to mobile entities is based on a human notion of trust [Jen02].

2.1.2 Mobile network models

Wireless networks may adopt one of two models for communication [CWKS97], which are differentiated by the level of infrastructure deployed in the environment. As [Haa03] notes, the two models are not mutually exclusive and a given environment may contain both types.

The infrastructure model

In the *infrastructure* network model, a set of stationary *access points* co-ordinate communications between mobile devices and provide gateway access to a fixed network. Access points have both fixed and wireless network interfaces, and to connect to the network a mobile device has to be within transmission range of an access point. This requirement imposes a severe restriction on the infrastructure model, since it may only operate in close proximity (hundreds of meters at best) to fixed infrastructure.

The ad hoc model

In the *ad hoc* network model, the network is composed only of a set of mobile nodes interconnected by wireless links, which may move randomly leading to rapid and unpredictable changes to the network topology. Perkins et al. define an ad hoc network as "the co-operative engagement of a collection of mobile nodes without the required intervention of any centralised

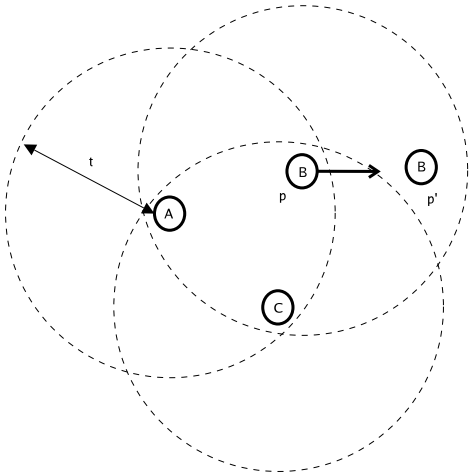

Fig. 2.1: A mobile, ad hoc network consisting of three nodes

access point or existing infrastructure" [PR99]. Ad hoc networks are becoming popular due to the ease with which they may be deployed, as well as the flexibility they offer in contrast to the overhead of setting up traditional fixed networks. Such networks are particularly attractive in situations where fixed infrastructure is not deployed, or has been destroyed, and communication ability is required rapidly, e.g., a disaster area or war zone. The ad hoc network model is vital to the realisation of pervasive computing where multitudes of mobile devices interact with each other in a dynamic and unpredictable manner in the absence of costly fixed infrastructure.

Each mobile node in a mobile ad hoc network (MANET) can combine the functionality of a router and a host, forming the network routing infrastructure in an ad hoc manner, or simply share a common broadcast region in a limited spatial area. The union of nodes forms an arbitrary graph in which nodes may move randomly. Fixed and infrastructure-based wireless networks use protocols that leverage their relatively static network topology and the fact that links between nodes in the network are reliable. Such assumptions do not hold in ad hoc networks and result in the following characteristics:

1. Network partitions - as a result of rapid and unpredictable mobility, *partitions* can occur frequently in the network, whereby the network is split into a set of disconnected

portions. For example, a mobile ad hoc network consisting of 3 nodes, each with transmission range t is illustrated in Figure 2.1. It can be seen that if node B, at position p continues to move in the direction indicated by the arrow $\rightarrow$, when it reaches position p', it will be out of range of the other nodes in the network and will be partitioned from them. Network partitions can cause severe disruption to network routing if they are not merged rapidly, which in turn affects higher level applications.

2. Routing - most routing protocols, designed for networks with infrequent topology changes, rely on the proactive exchange of topology information between nodes and the use of routing algorithms to inexpensively compute routes through the network. However, in a MANET, where the topology changes constantly and bandwidth, power, and transmission range are constrained, traditional routing protocols do not perform well and both reactive [PR99, JM96] and proactive [CP94] ad hoc routing protocols have been proposed.

We argue that the ad hoc network model is of particular value to the developers of mobile, context-aware applications in pervasive environments, where application components may collaborate anywhere, potentially in the absence of any fixed network infrastructure. The characteristics of mobile, ad hoc networks may also be used by the application developer to react to contextual events such as an impending network partition.

2.2 Context-aware computing

The emergence of mobile computing has given rise to applications which can exist in a range of different environments or *contexts*, during execution. In contrast to stationary computing, an application on a mobile device may experience rapidly changing physical conditions such as location, bandwidth availability and economy, and logical conditions such as administrative domain. The increasing availability of a wide range of diverse sensors allow applications to be aware of further environmental parameters such as velocity, orientation, temperature and a host of others, leading to a new class of application which is able to sense and adapt to its context.

2.2.1 Definition of context

Although research into context-aware computing was performed as far back as 1992 with the Olivetti Active Badge project [WHFG92], the first work to introduce the term *context-aware* was that by Schilit and Theimer [ST94], with their work on active maps. This work takes a user-centric approach to context and defines context-aware computing as 'the ability of a mobile user's applications to discover and react to changes in the environment they are situated in' [ST94]. This definition of context includes the computing, user and physical environment. A later refinement of this definition lists the three important aspects of context as being (1) where you are; (2) who you are with; and (3) what resources are nearby [SAW94].

Brown [BBC97] and Castro [CCKM01] both define context as being information about location, the identity of people in close proximity, physical conditions, and accessible resources, whilst Pascoe [Pas98] gives a broad definition of context as the subset of physical and conceptual states of interest to a particular entity. Ryan et al. introduce the notion of time in their definition of context which encompasses information about location, time, identity and physical environment [RPM97]. Hull et al. [HNBR97] define context in a similar manner to be aspects of the *situation* of an application including identity, location, companions, computing resources and physical environment. The use of situation goes beyond the capture of physical conditions. This is reinforced by Dix et al. [DRD+00] who provide a detailed treatment of context which includes *infrastructure context*, *system context*, *domain context* and *physical context*. This definition considers the nature of the application itself (e.g., pace and interaction), as well as semantics of the application domain (e.g., style of use) in addition to physical and infrastructural considerations. Gellersen et al. [GSB02] note that confusion can arise from use of the term context at different levels of abstraction to denote either the real world situation of an application, or one aspect of the situation such as location, or a specific instance of some aspect, such as a place. Their definition of context is information can be acquired from the real world through sensor capture and fusion, as opposed to situation which is what information exists in the real world and is a slightly more restrictive definition than [DRD+00].

A hierarchical approach to context is proposed by Schmidt et al. [SB98] who structure

context using the model that (1) a context describes a situation and the environment a device is in; (2) a context is identified by a unique name; (3) for each context a set of features is relevant; and (4) for each feature a range of values is determined by the context. They [SB98] define a hierarchy for context data with the top level consisting of the two broad categories of *human factors* and *physical environment* and providing three sub categories to each of these. It is argued that this model provides structure for dealing with context.

Perhaps one of the most popular definitions of context currently in use is that of Dey et al. [DA99] who define context as *"any information that can be used to characterize the situation of an entity, where an entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and the application themselves"*. This definition is intended to ease the enumeration of context for a given application scenario by an application developer, but the definition is in danger of being too broad. As Winograd [Win01] points out, the term *any information* means just that and could encompass any information from that about the electric power grid to system files used in compilation. Dey et al. elaborate their definition of context by providing four categories of context that they feel are more practically important than others. These are **location**, **identity**, **activity** and **time** and are equivalent to those proposed by Ryan et al. [RPM97], except that **activity** replaces **environment** in Ryan et al.'s definition.

Chen and Kotz [CK00] differentiate between environmental information that *determines* the behaviour of mobile applications and that which is *relevant* to the application in their definition of context as 'the set of environmental states and settings that either determines an application's behaviour or in which an application event occurs and is interesting to the user'.

2.2.2 Definition of context-awareness

Schilit and Theimer provide the first definition of *context-aware computing* in the literature as 'the ability of a mobile user's applications to discover and react to changes in the environment they are situated in' [ST94]. Schilit et al. go on to describe four categories of context-aware applications, as illustrated in Table 2.2. These categories are positioned along two orthogonal

	manual	**automatic**
information	proximate selection + contextual information	automatic contextual reconfiguration
command	contextual commands	context-triggered actions

Table 2.2: Categories of context-aware applications [SAW94]

dimensions representing whether (1) the task is the retrieval of information or the carrying out of a command; and (2) whether it is carried out automatically or manually.

This classification yields the following four categories of context-aware applications:

1. *Proximate selection* is a technique used in user-interfaces whereby objects located nearby a user are emphasised.

2. *Automatic contextual reconfiguration* refers to the process of adding or removing components or the connections between them due to changes in an application's context.

3. *Contextual commands* are commands whose execution is modified based on current context data.

4. *Context-triggered actions* are actions which are executed automatically when a certain context exists and are based on if-then rules.

Brown et al. [BBC97] define context-aware applications as those that change their behaviour according to the user's context. In some respects this is a broader definition than that of Schilit et al. since it does not deal with the discovery or acquisition of context and does not explicitly mention mobility. This definition is also restricted to adaptation to the context of the *user*. Since user and application may be physically or logically distributed, this condition may be overly restrictive.

Pascoe defines context-awareness as the ability of a program or device to sense various states of its environment and itself [Pas98]. Implicit in this definition is the fact that the program or device uses the sensed context in some way during the course of its execution. Whilst Schilit et al. identify classes of context-aware application, Pascoe proposes a set of

four features of context-aware applications, or core generic capabilities, which can be used as a vocabulary to identify and describe context-awareness independently of application, function, or interface [Pas98]. This first of these features is *contextual sensing* which refers to the detection of environmental states and their presentation to the user. This is similar to the proximate selection class of context-aware application defined by Schilit et al. *Contextual adaptation* refers to the adaptation of application behaviour to the current context and is similar to Schilit et al.'s context triggered action class of application. *Contextual resource discovery* is the use of context data to discover other resources within the same context. This category bears some resemblance to Schilit et al.'s automatic contextual reconfiguration class of application, except that Schilit et al.'s definition does not take into account awareness of the contexts of other entities. The fourth and final capability defined is that of *contextual augmentation* in which the environment is augmented with digital data associated to a particular context.

Dey and Abowd [DA99] propose a categorization of the features of context-aware applications that combines the ideas of Schilit and Pascoe whilst addressing the differences between the two. They consequently define three categories of features that context-aware applications may support as (1) *presentation* of information and services to the user; (2) automatic *execution* of a service; and (3) *tagging* of context to information for later retrieval. The definition of *context-aware* given by Dey et al. is 'the use of context to provide relevant information and/or services to the user, where relevancy depends on the user's task' [DA99].

Chen et al. divide context-aware computing into two broad categories in their definition [CK00]. *Active context-awareness* refers to the automatic adaptation of application behaviour according to context, whilst *passive context-awareness* refers to the storage and presentation of context data to the user.

2.2.3 A working definition of context

We note that the majority of existing definitions of context in the literature take a user interaction-centric view of context as being information that influences the way in which a user interacts with an application. The commonly accepted vision of pervasive computing as

'disappearing' into the background of everyday use [Wei91] suggests that a continuing reliance on explicit user interaction can only serve to hinder the realisation of this vision. We argue that the pervasive computing function should necessarily be autonomous and proactive, and therefore propose a definition of context that is less centered in user interaction.

We define *context* as environmental information that an application may use to autonomously and proactively fulfill its goals. This definition encompasses the fact that context determines the state of the application and constrains the set of useful behaviours and actions at a particular point in time. *Environmental* information in this regard means both information from the physical environment, as well as infrastructural information from the underlying execution environment of an application. The *type* of environmental information of interest is influenced by Dey et al.'s categorization [DA99]. It is our belief that this definition of context, being less user-centric than previous definitions, recognises the requirement for pervasive computing not to rely on explicit human-system interaction. Our definition of *context-awareness* follows as the use of context information by an application in the fulfillment of its goals.

2.2.4 The role of proximity

The central role played by location in context-aware applications today is due in part to the fact that mobility has made location an important variable, the role of which is well understood, and in part to the fact that a wide range of location-sensing technologies are fairly mature and accessible, e.g, GPS [HWLC94], GSM [Wil98], ultrasonic positioning systems [WJH97], and positioning using IEEE 802.11 wireless networks [CCKM01]. Context-awareness benefits both from the notion of *absolute* location, which is exact location described by a co-ordinate system, and *relative* location, which is location with regard to some reference point, both of which in turn benefit from a notion of *proximity*, which may be defined as the property of being close together. Proximity is important in context-aware applications since many of these applications make use of spatial locality in behaviour adaptation, e.g., [WJH97, CMD99, AAJ+97], and most definitions of context include some notion of proximity [BBC97, SAW94, GSB02]. The notion of proximity is not necessarily constrained to spatial

proximity, but includes *logical* proximity where entities may be physically remote, but close together in some other way, e.g., administrative domain.

2.2.5 Challenges to developing context-aware applications

Whilst the incorporation of context data into applications to make them context-aware is key to the realisation of pervasive computing as is the disappearance of the computing function into the background, building mobile, context-aware applications remains a challenging undertaking. This is due to the fact that at present, application developers are required to develop, from scratch, software to capture, represent, and process context data in a mobile environment. There is no commonly-accepted programming model promoting scalability, extensibility, and reuse of application components, and most importantly, ease of development.

In this section we discuss the major challenges to developing context-aware applications in mobile environments that must be addressed by a programming model.

Capture of context data

In addition to identifying the relevant sources of context data for a particular application, the application developer often has to write low-level code to interact with sensor hardware at device protocol level, e.g., [SF99, RJD99]. Such development is time-consuming, error-prone, and only accessible to fairly experienced programmers, and whilst a number of approaches to developing context-aware applications define abstractions to assist in context capture [DSA01, CM00, Hon00, SAT+99], most do not provide support to the developer for the representation and processing of context data.

A usable abstraction for dealing with the capture of context data from low-level sensor hardware is essential for developers of mobile, context-aware applications, as is the incorporation of the abstraction into an overall programming model supporting the representation and processing of context data.

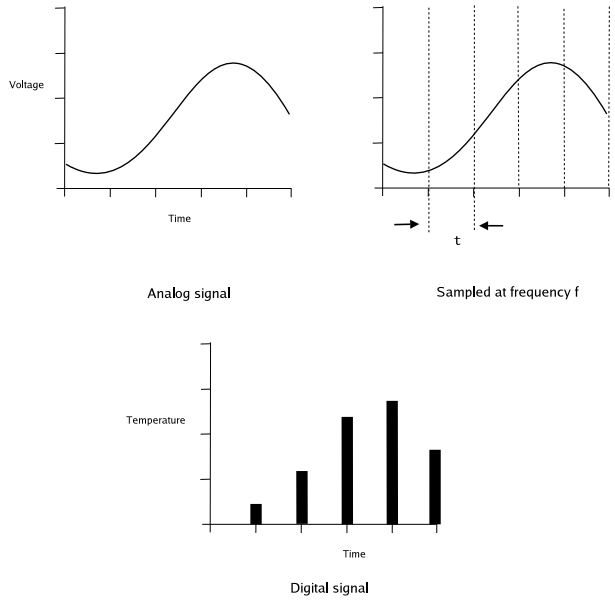

Fig. 2.2: Quantisation of an analog signal to a digital signal

Uncertainty of context data

Measurements made of the real world by sensors based on physical transducers will always contain a degree of *uncertainty* and *incompleteness*, which together result in an inherent unreliability of context data based on such measurements. Uncertainty regarding the true value of what the sensor is measuring is inherent in data resulting from a physical measurement and stems from hardware limitations in the manufacturing of the sensor and the fact that the physical operation of the sensor is too complex to model. Hardware sensors typically produce a continuous time, continuous amplitude *analog* signal, with infinite precision. In order to use this analog signal in a computer, it has to be converted into a digital signal, in a process known as *quantisation* whereby the state is constrained to a set of discrete values, rather than varying continuously. A *digital* signal is thus a discrete-time, discrete amplitude signal defined only at sampling times with finite precision. Figure 2.2 illustrates the process of analog to digital signal processing. A continuously varying voltage output by a temperature sensor is sampled at intervals of t, to produce a digital signal which is an approximation

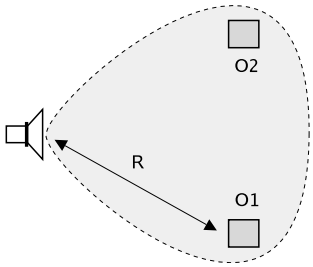

Fig. 2.3: Inherent uncertainty in an ultrasonic range finder reading

of the signal by a set of discrete temperature readings. It is easy to see how the process of analog to digital conversion involves the systematic loss of data, since the conversion process only has a finite resolution. This quantisation error is one source of uncertainty in sensor data, with others arising from measurement errors made by the sensor.

A classic example of the uncertainty inherent in a sensor reading is given by [Vis99] for an ultrasonic range finding sensor, as illustrated in Figure 2.3. This type of sensor can detect the distance to an obstacle within its 'cone' of vision, but is not able to determine the position of the obstacle. In the figure, the sensor would not be able to discriminate between the position of obstacle $O1$ and obstacle $O2$ - the range value R will be the same for both obstacles. In addition to the inherent uncertainty of sensor data, each type of sensor performs a narrow and specific sensing task and is unable to capture completely all aspects of a particular context. For example, for the sensor illustrated in Figure 2.3, if one obstacle lies slightly closer to the sensor than the other, the sensor will only detect the nearest obstacle.

It is important to provide systematic support for dealing with uncertainty to developers of context-aware applications. Whilst numerous approaches have been proposed for context-aware applications [WSSY02, WSA02, RAMC04, CM00, DMAC02, CSG99] there remains no commonly accepted and generic approach that is part of an overall programming model, with most developers rather managing uncertainty in an ad hoc and application-specific manner.

Representation of context data

In order to process, reason about, and react to context data, a systematic approach to the representation of context data is required by the application developer. The selected representation format should be efficient to process and reason about by the application. A variety of approaches to representing context data have been proposed [Win01, Rya99, FVB02, HIR02, RC03b, Bro96, SAT+99, Hon00, RMCM03, dInK01], but no commonly accepted approach has emerged.

Developers of context-aware applications need a systematic and powerful approach to the representation of context data within applications, in order to efficiently make useful inferences on this data. Furthermore, the representation format should be closely integrated with the inference process, promoting efficiency.

Privacy

Traditional concerns regarding privacy are amplified in context-aware applications which are predicated on access to a wide range of sensitive data, and involve ad hoc collaborations between entities. Context-aware computing connotes the storage of more data, with the associated increased risk of theft and misuse.

The explicit incorporation of location, activity, and identity data into applications raises serious privacy concerns [Lan02] which have been voiced right from early applications [Coy92]. If context-aware computing is to be embraced as a mainstream technology, privacy of sensitive data has to be assured, and application developers require appropriate tools to manage privacy and security. Although approaches to managing privacy in context-aware applications have been proposed [OR03, Can02], there remains no common approach to managing these concerns as part of an overall programming model.

Scalability

Scalability refers to the ability to incrementally increase the abilities of a system, whilst maintaining, or improving, performance. Context-aware applications in mobile ad hoc environments will form a part of an overall pervasive computing infrastructure consisting of very

large and dynamic distributed populations of entities, and thus scalability of communication is an important consideration to application developers.

Scalability is a significant challenge in the mobile ad hoc networks we envisage as being crucial to pervasive computing environments, due to the large increase in the network protocol control overhead experienced with an increase in the number of nodes in the network [LBC+01]. Within such an environment, the provisioning of QoS is a significant challenge, and a number of proposals have been made to address the characteristics of such networks [ACVS02, LAZC00, XTB+03].

It is essential that developers of context-aware applications in mobile environments have appropriate abstractions and system support available to ensure the scalability, and ubiquitous adoption, of their applications.

Synchrony

Most existing distributed applications are based on *synchronous* operation whereby an operation has to wait for a response before execution can continue. This method of operation is not entirely adequate for context-aware applications which need to be notified *asynchronously* when new context data is available. Synchronous operations in context-aware applications imply expensive polling behaviour in order to determine when the requisite information is available [BMB+00]. Such expensive communication behaviour is not suited to the resource constraints inherent in mobile devices.

Support for both synchronous and asynchronous communication is important for the developers of mobile, context-aware applications, although most current approaches to context-aware application development are tightly coupled client-server architectures based exclusively on synchronous invocations using mechanisms such as HTTP [DSA01] and CORBA [RC03a], and their remains poor programmer support for developing context-aware applications based on asynchronous communication.

Extensibility and reusability

Extensibility may be defined as the ability to add new functionality to an application, whilst reusability may be defined as the ability of a piece of functionality to be used again, unmodified, in a different system than it was originally written for [Eng97]. It is likely that in the future multiple, unanticipated types and sources of context will become available, whilst new applications will emerge that use existing sources of context. The ability to seamlessly integrate new sources of context data into applications, whilst at the same time re-using existing functionality is essential to the realisation of pervasive computing. Current approaches to context-aware application development with ad hoc integration of devices and application logic results in applications that are neither extensible, nor reusable.

Support for extensibility and reusability is an essential requirement of a programming model for context-aware applications. Facilitating extensibility and reusability of application components, enables the incremental evolution of applications, reducing development effort and reducing the need to develop from scratch.

Summary

The fundamental challenge to the development of mobile, context-aware applications remains the lack of a widely accepted, domain-specific programming model, providing support for the challenges discussed above. Whilst extensive support exists for addressing individual challenges such as the capture of context data [DSA01, SAT+99], the management of uncertain context data [CM00, RAMC04], the efficient representation of context data [GA99, RMCM03], the management of privacy and trust [KFJ01, OR03], and scalable asynchronous communication in mobile, ad hoc networks [MC03], there is no unifying model providing this support to the programmer in an easily accessible manner to enable the development of context-aware applications.

2.3 Supporting development of mobile, context-aware systems

The previous section has discussed the major challenges faced by developers of context-aware applications in mobile ad hoc environments. In this section, we identify a set of requirements based upon these challenges, which we believe are necessary in providing domain-specific support to the developers of mobile, context-aware applications. We acknowledge the criticality of privacy and security within such environments, but scope our research to specifically exclude privacy and security considerations at this point. Privacy and security are excluded at this point, since they constitute a vast research area in and of themselves, with separate projects dealing exclusively with this area being undertaken, e.g, the SECURE project [CGS+03]. It is envisaged that support for security and privacy will be incorporated into the programming model at a latter date.

2.3.1 Loosely coupled communication

The communication paradigm adopted by context-aware applications in mobile, ad hoc environments as envisaged in pervasive computing scenarios should be dynamic, supporting the frequent mobility and unpredictable interaction patterns characteristic of such networks. Applications operating in such environments cannot rely on traditional distributed computing communication paradigms where the sender of a message knows the identity of the intended recipient a priori. Traditional methods of communication based on point-to-point, request/reply models are infeasible since (1) the address of all interacting entities has to be known a priori; and (2) this paradigm only supports one-to-one communication semantics and does not scale well to the large numbers of entities envisioned in pervasive environments.

An anonymous, generative event-based communication paradigm is well suited to mobile ad hoc environments. This type of communication paradigm is anonymous since an entity producing an event need not know which entities (if any) have subscribed to the type of event and will thus receive it. The anonymity and many-to-many style of event-based communication addresses both scalability and extensibility issues.

Event-based communication

The event-based communication paradigm provides anonymous, loosely coupled, many-to-many communication between application components via asynchronous event notifications [BMB+00]. *Event notifications* represent a change in the state of the sending application component, and are propagated from *producers* (sending application components), to *consumers* according to subscriptions made by consumers [MC02a]. Events typically have a name and a set of typed attributes, and *event filters* provide a mechanism to scope the delivery of events to consumers based on declared interest.

Event-based communication in an ad hoc wireless environment poses additional challenges since the event middleware cannot rely on the presence of access points to route messages, nor can it rely on intermediate components which may apply event filters or enforce non-functional attributes [MC03].

It is crucial to provide developers of context-aware applications in mobile, ad hoc environments with both system support for loosely coupled communication in mobile, ad hoc environments, as well as tool support for developing applications based on this communication paradigm.

2.3.2 Sensor abstraction

A number of enabling technologies have contributed to the rise of cheap, ubiquitous and high-performance sensors [Saf97]. Among these technologies are MicroElectroMechanical Systems (MEMS), piezo materials, charge-coupled devices (CCD), and at a higher level, Global Positioning System (GPS) [HWLC94] satellites for location sensing.

A sensor is defined as a device that responds to some form of physical stimuli (such as a change in temperature), by producing an electrical signal. As such, a sensor is essentially a transducer, a component which converts one type of energy to another. For example, a temperature sensor may convert a change in physical temperature to an analog electric signal, such as a varying voltage. In addition to the traditional definition of a sensor as responding to *physical* stimuli, context-aware applications often depend on components which respond to digital stimuli from software rather than the physical environment, e.g, a web service that

reports estimated journey time between two towns.

Hardware sensors usually produce numerical output using low-level, device-specific protocols (e.g., see sections 6.1.5 and 6.2.1). Integrating the output of sensors into applications typically requires significant low-level knowledge, and often results in tightly coupled applications and limited reusability. Crucial in easing the development of context-aware applications is the provision of software components which abstract away from physical device protocols, and support the conversion of numerical protocols into a higher-level symbolic representation. Few application-level developers have experience of working with low-level hardware and to ensure the development process is accessible to as wide an audience as possible, it is essential that some way of abstracting away from individual devices is provided.

2.3.3 Sensor fusion

As discussed in section 2.2.5, sensor data obtained from a transducer is inherently unreliable primarily due to the conversion of a continuous time, continuous amplitude analog signal with infinite precision, to a digital signal with finite precision, within the sensor hardware. Whilst expensive sensors may offer a high degree of reliability, by definition pervasive computing implies the adoption of inexpensive sensors, whilst requiring resolution and accuracy commensurate with human perceptive ability [WSA02]. A scheme which manages the unreliability of inexpensive sensors is consequently an essential requirement of a programming model for context-aware applications in pervasive environments. A proven approach to managing sensor uncertainty is the combination of readings from multiple sensors, or multiple readings from the same sensor. This technique is known as *sensor fusion*[4], and allows inferences to be made that might not be possible from a single reading from a single sensor. In general, there are two broad categories of sensor fusion.

Mono-modal sensor fusion

The potential uncertainty present in a single reading produced by a single sensor may be reduced by fusing multiple readings from the same sensor at different points in time, using

[4]Alternately referred to in the literature as *data fusion* and *information fusion*

techniques such as the Kalman filter [Kal60]. This provides a more accurate description of the measured parameter than a single reading and may be applied by using sensor readings to successively update the estimation of the parameter being measured.

The uncertainty of readings from an individual sensor may further be reduced by fusing the output of a set of redundant sensors measuring the same parameter at the same point in time, using numerical techniques such as sum and average. Mono-modal sensor fusion reduces the uncertainty of sensor data and increases its accuracy.

Multi-modal sensor fusion

The incompleteness of sensor data may be mitigated by fusing the output of several disparate sensors measuring different environmental parameters in a complementary approach known as *multi-modal* sensor fusion.

Whilst the majority of approaches to sensor fusion deal with fusing the output of multiple sensors of a similar type, fusing sensory output of different modalities is a substantially more difficult task. Context-aware systems typically rely on a wide range and type of sensors in order to accurately derive their context, so for example, may need to fuse the output of a PIR (passive infrared) sensor, a pressure sensor, and a light sensor to determine the action currently taking place within an office. Mono-modal techniques which exploit the similarity of their inputs, extracting features and merging these together, are not applicable. The difficulties inherent in fusing sensor readings of different modalities has led to most solutions being highly application-specific and not extensible beyond a specific set of sensors and a specific task. A number of approaches to fusing multi-modal sensor data in context-aware applications have been proposed, including rule-based approaches [SAT+99, dInK01], Dempster-Schafer Theory [WSSY02], and probabilistic networks [CM00, RAMC04], but until now these have been tightly integrated with specific applications.

It is essential to provide application developers with a generic yet systematic approach to managing the uncertainty inherent in a single sensor reading. In order to support the development of future systems, a generic approach has to be provided that is applicable across a range of applications, whilst ensuring accessibility to a range of developers.

2.3.4 Context representation

Context data obtained from sensors needs to be represented and stored within the application using some data structure. Chen and Kotz [CK00] argue that although most existing applications use ad hoc data structures to represent context data, they typically fall into one of the following broad categories.

Key-value pairs

Context data may be represented as a set of key-value pairs, as in pioneering work by Schilit at al. in [STW93], where the key represents an environmental variable of interest to the application, and the value its current value. An example set of key-value pairs representing a geographical location presented as a latitude/longitude pair is illustrated below:

```
latitude='1750.0000'
longitude='3130.0000'
```

This approach provides a simple and extensible approach to representing context data, but is unstructured and attaches no semantics to the data represented.

Tagged encoding

This approach models context data as Standard Generalised Markup Language (SGML) [fS86] documents containing tags and corresponding fields. One approach to the representation of context data using tagged encoding, is the use of an application-specific schema.

The disadvantage of developing application-specific schemas is that they are not accepted as standard and thus are not interoperable with other systems. Another approach is to use a standardised schema, such as the resource description framework (RDF) [MSB04], developed by the W3C. RDF is an XML-based foundation for describing metadata used for semantic knowledge modelling and provides a simple data model for describing resources. A *resource* is an object that can be referenced and is identifiable by a URI. Resources have associated *properties* which in turn have a *value*. A value may itself be atomic, or may be another resource, which in turn has its own properties.

Whilst the primary application of RDF has been as a universal data format for the Web used to describe web resources such as HTML pages and graphics, it has been applied to the representation of real-world context data [FVB02]. An RDF fragment describing the location of a resource is illustrated below.

```
<rdf:RDF
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:location="http://www.dsg.cs.tcd.ie/location">
 <rdf:Description about="http://www.dsg.cs.tcd.ie/~biegelg">
  <location:latitude>1750.0000</location:latitude>
  <location:longitude>3130.0000</location:longitude>
 </rdf:Description>
</rdf:RDF>
```

Composite capabilities preference profiles (CC/PP) [BHK04] is another standard developed by the W3C to support content negotiation between browsers and servers and is based on RDF. CC/PP currently provides a model and vocabulary for describing device capabilities and user preferences for mobile devices, but there has been work carried out to incorporate other types of context data [IRRH03, OR02].

The major advantage of an approach to context representation based on tagged encoding is that it is extensible and can be used to describe virtually any context data. Furthermore, ontological approaches to representation permit the association of semantics with the data. A disadvantage of this approach is that the verbosity of the representation format is not well suited to resource constrained environments, and the data is not easily reasoned about.

Object-oriented

Context data may be represented as a set of objects encapsulating variables, and associated accessors and mutators. A class describing an object which represents location context data as a pair of variables representing longitude and latitude is illustrated below:

```
class Location{
  private double latitude, longitude;
  public double getLatitude(){}
  public double getLongitude(){}
  public void setLatitude(double newLatitude){}
  public void setLongitude(double new Longitude){}
}
```

An object oriented approach to context representation is adopted in the GUIDE project [CMD99] where a *position sensor* object represents location context data based on signals received from remote base stations, as well in [HHS+99] to model real-world objects in a sentient application. This approach has the features associated with object-orientation, namely inheritance, encapsulation, and polymorphism.

Logic-based

Following this approach, context data is expressed as a set of facts in the working memory of a rule-based system. By storing context data as facts directly within the rule-based system, context data is closely coupled with the rules that perform inference based on it. This approach is successfully adopted in [dInK01] to store sensor data. An example of a fact, representing a fragment of context data, being asserted in a knowledge base is illustrated below:

```
(assert (location (latitude '1750.0000') (longitude '3130.0000')))
```

The greatest advantage of this approach to context representation is that the context data may efficiently be reasoned about since it is stored directly in the knowledge base of a rule-based system, which is used by an inference engine to make decisions.

It is vital to provide developers of context-aware applications with a systematic and structured approach to the representation of context data within applications. The selected representation format should be integrated with the inference mechanism to allow the application to reason efficiently about context data.

2.3.5 Inference

Context-aware systems perform actions based on context data derived from sensor inputs. This requires the system to reason from observations made by sensors, to conclusions in a process known as *inference*. Whilst there are a wide range of possible approaches to providing inference capabilities to context-aware applications, rule-based systems and machine-learning approaches have emerged as the major contenders.

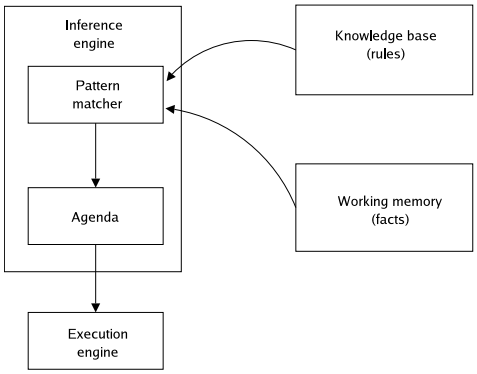

Fig. 2.4: Architecture of a rule based system adapted from [FH03]

Rule-based systems

Rule-based systems provide one approach to inference which is widely adopted amongst context-aware systems [dInK01, RC03a]. In such systems, the reasoning process uses a set of *facts*, and knowledge captured as *rules* applied to these facts to draw conclusions, given a set of observations. For example, from a very early age humans use the observation that someone is crying, combined with rules learned by experience, to infer that the person is unhappy. The certainty of an inference is based on the quality of both the observation and the underlying rules.

Rule-based systems are programmed declaratively, i.e., the programmer specifies a set of conditions and actions, leaving it to the system to work out how to fulfill them - the order in which the logic is specified is not important. Declarative programming provides a higher level of abstraction than procedural programming and is more flexible when inputs are incomplete or poorly specified. A typical rule-based system is composed of the 3 major components illustrated in Figure 2.4.

1. The **knowledge base** contains the set of rules against which make inferences are made.
2. The **working memory** contains a set of *facts* representing the fragments of information with which the system is currently working.
3. The **inference engine** implements algorithms that apply rules contained in the knowl-

edge base to facts present in the working memory. Within the inference engine a *pattern matcher* matches rules with facts, to determine which rules are fired to form the *agenda*, the set of rules which will be fired.

Rules-based systems are programmed by specifying the knowledge base through a set of rules. Facts are then supplied to the system, either by the programmer or by the application, and the inference engine provides output after applying the rules to the facts.

Machine learning

Machine learning refers to the use of a set of algorithms to infer a model from a set of data. In terms of inference in context-aware applications, machine learning algorithms are of interest both in the classification of contexts from noisy sensor data, and in the learning of appropriate behaviour in different contexts, rather than relying on behavioural rules specified by developers. Although not yet widely employed as an inference mechanism in context-aware applications, some machine learning algorithms have been adopted.

1. Naïve Bayes classifier - this is a simple classification method based on a probability model derived from Bayes' Theorem [Bay58], and making strong independence assumptions. The use of Naïve Bayes classifiers has previously been proposed for deriving context classifications from sensor data, e.g., [RC03a].

2. Reinforcement learning - broadly speaking, reinforcement learning algorithms map situations into actions guided by trial and error. According to [KLM96], in the standard reinforcement learning model, an entity is connected to it's environment via perception and action. At each interaction step the agent receives input i as indication of the state s of the environment, and generates an action a as output. The action changes the state of the environment, communicated to the entity by reinforcement signal r. The entity then chooses behaviours which increase the sum of values of r in the long term [KLM96].

3. Artificial neural networks - artificial neural networks are parallel computing devices consisting of many interconnected simple processors (nodes) [Cal03]. Interconnections

between the nodes make up a large part of the intelligence of the network, and the network has to be trained to enable useful computation to take place. Artificial neural networks have previously been applied in context-aware applications, e.g., [Moz98].

Machine learning has been proposed as a more flexible approach to inference in context-aware applications, allowing applications to 'learn' behaviours in different contexts, rather than following a rigid rule-set defined by an application developer.

Developers of context-aware applications should be provided with a structured means to reason about context data. None of the inference mechanisms described are readily accessible to average developers due to their relatively complex programming models, and higher-level support is necessary in order to offer this functionality to developers.

2.3.6 Actuator abstraction

Actuators provide a useful abstraction for talking about the actions taken by context-aware applications. A traditional definition of an *actuator* is a device which responds to an electrical signal by producing a mechanical action, such as motion, or acoustic or thermal energy. This fairly narrow definition constrains actuation to effecting a change in the physical environment and in its current form is not adequate for context-aware applications, since not all applications' actions effect a change in the physical environment. Many context-aware applications only perform actions that effect a change in software, e.g., customising a Graphical User Interface (GUI), and this needs to be taken into account when considering actuation.

Interaction with most hardware actuator devices is via low-level, device-specific protocols, whilst interaction with software actuator devices is via custom APIs. Programming actuator devices is a complex task, which is only available to experienced developers with experience of either the hardware or relevant API. It is thus essential that any approach to supporting the development of context-aware applications provides an appropriate abstraction for interacting with actuator devices. The major function of such an actuator abstraction is the conversion of high-level, symbolic commands, into low-level commands based on numerical, device-specific protocols.

2.3.7 Developer support

Consolidating the set of other components, developer support is required in the form of a programming environment which exposes the components to the application developer in a coherent and easily accessible manner. Support has been offered to developers to some degree for a subset of the requirements discussed above, e.g., [DS03, DSA01, RC03a, SAT+99] but often this support is inaccessible to all but the most experienced of developers, or does not provide comprehensive support for all the requirements identified in the preceding sections.

Domain specific languages A domain specific language (DSL) is a programming language which is closely related to a particular problem domain, in contrast to general-purpose programming languages that may be applied to range of problem domains. DSLs are characterised by high-level, domain-specific constructs and have very specific goals in their design and implementation. The rationale behind DSLs is a reduction of application development complexity through the provision of high-level, domain-specific abstractions, easing the specification of application functionality. DSLs are not a new concept, having been proposed and discussed since the beginning of computing [Lan66], and provide a promising approach to providing support to developers of mobile, context-aware applications.

An excellent overview of the commonly accepted advantages and disadvantages of DSLs is provided by [vDKV00], with the major advantages of DSLs with regard to the provision of developer support summarised from [vDKV00] as follows:

- DSLs embody domain knowledge, enabling the conservation and re-use of this knowledge by application, without the need for developers to explicitly code it into each application
- DSLs enhance developer productivity, as well as increasing the reliability, maintainability, and portability of application code
- DSLs allow applications to be specified in terms of domain abstractions, making application development more accessible to domain experts who are generally not skilled programmers

The major general disadvantages of DSLs summarised from [vDKV00] are as follows:

- The major initial cost associated with the design, implementation, and maintenance, of a DSL
- The need to train developers in the use of a DSL
- The potential for loss of efficiency when compared to hand-coded software that may provide for low-level, application-specific optimisations

The major advantage provided by a domain specific language to developers of mobile, context-aware applications is abstraction away from low-level complexities through domain-specific constructs, making application development accessible to a much wider audience. A wider audience in this regard is considered as experienced computer users, who do not necessarily have significant exposure to low-level programming. For example, the range of users who have experience of composing formulae in a spreadsheet application, is significantly wider than those with experience of opening a socket and specifying a network protocol, or designing and implementing an object-oriented application. It is such accessibility to a greater majority of people that we believe is crucial in the widespread deployment of context-aware applications in the realisation of truly pervasive computing.

Code generation Many domain specific languages include a code generation step, the function of which is to transform a domain-specific specification into a lower-level target language, such as Java. The advantages of code generation from a domain-specific model include the speed of generation, and the consistency and stability of the generated code. Most importantly, the provision of high-level, domain-specific abstractions which are subsequently transformed to low-level code removes the need for application developers to develop low-level implementations themselves - a time-consuming and error-prone process, accessible only to skilled developers. Important disadvantages related to automatic code generation that need to be borne in mind include the notable disadvantage that the initial development of the code generator requires substantial, complex, development effort, and there will always be some code that cannot be automatically generated and must be hand-crafted.

In order for Weiser's vision of pervasive computing to be fully realised, application development has to be made accessible to as wide a range of potential developers as possible. Domain specific languages with their associated target language code generators are a proven approach to providing high-level support to application developers within a specific domain. Such support typically makes application development significantly more widely accessible to those without explicit low-level programming experience.

2.3.8 Requirements

Based on our discussion of the challenges faced by the developers of mobile, context-aware applications, the set of requirements we have derived as essential in the development of a programming model for such applications is summarised below:

- **R1: Loosely coupled communication** - the programming model should support the development of application components that communicate using a loosely coupled communication mechanism that addresses mobility, as well as application scalability and extensibility.
- **R2: Sensor abstraction** - the programming model should provide a suitable high-level abstraction to facilitate the incorporation of sensor data from a range of sensing technologies, implemented both in hardware and software.
- **R3: Sensor fusion** - the programming model should provide a systematic and efficient approach to fusing the output of potentially multi-modal sensors as a way of mitigating the uncertainty of individual sensor readings in a timely manner. The approach should be generic, i.e., applicable to a wide range of potential application scenarios, whilst at the same time remaining accessible and usable.
- **R4: Context representation** - the programming model should provide an effective means to represent context information within the application. Given the potentially large volumes of data, efficiency should be emphasised in the approach.

- **R5: Inference engine** - a systematic and efficient approach to reasoning about context data should be provided by the programming model at a high level of abstraction.
- **R6: Actuator abstraction** - the programming model should provide a suitable abstraction for developers to specify interaction with the environment via a range of actuator devices, both hardware and software.
- **R7: Developer support** - an accessible and usable development environment should expose the support offered in the programming model, to the application developer.

2.4 Summary

This chapter began by introducing the mobile computing paradigm and discussing the challenges inherent therein, bringing forward the additional challenges posed by operation in an infrastructureless, ad hoc network. Context-aware computing was then introduced through examination of common definitions of context and context-awareness in the literature. A number of issues pertaining to context-aware computing were then considered, before a set of requirements considered to be crucial in supporting the development of context-aware applications in mobile ad hoc environments were postulated. Popular approaches to fulfilling some of the requirements were presented where they exist. Support for this set of requirements is considered crucial in the provision of generic and accessible support to the developer. In the next chapter we analyse state-of-the-art approaches to supporting the development of context-aware applications with regard to this set of requirements.

Chapter 3

State of the Art

This chapter examines state-of-the-art projects that each go some way to providing an approach to supporting the development of context-aware applications. We discuss the extent to which each of these existing projects provides support for the core requirements required to facilitate the development of context-aware applications in mobile, ad hoc environments, as identified in the preceding chapter.

Whilst a number of state-of-the-art projects exist that offer support for the development of context-aware applications, our conclusion is that no single approach offers support for the complete set of requirements needed by the developers of context-aware applications in mobile ad hoc environments.

3.1 Scope of this review

Pervasive computing research has now entered its second decade and as a result many projects have been undertaken that deal with a wide variety of topics within what is undoubtedly a very broad research area.

We thus direct our attention to the subset of pervasive computing research that focuses on providing generic support for the development of context-aware applications to some extent. In this review, we are particularly interested in the extent to which the research supports the requirements we identified in Section 2.3.

3.2 Stick-e note Architecture

The stick-e note software infrastructure developed at the University of Kent at Canterbury provides one of the first approaches to support the development of context-aware applications. The aim of the infrastructure is to significantly simplify the creation of context-aware applications using the electronic equivalent of a Post-it note [BBC97] and, as such, it focuses on information presentation and in particular *discrete* context-aware applications, i.e., those in which the information presented to the user does not change continuously. In such applications, separate pieces of information are attached to specific contexts (location, states, temporal ranges, adjacency) and are presented to the user when the appropriate context is entered. The infrastructure is aimed at mobile users carrying small computing devices, such as PDAs, enhanced with environmental sensors, but is essentially an off-line system and does not explicitly address mobility issues.

A *stick-e note* is created as an SGML [fS86] document that uses a set of SGML mark-up tags, defined in a Document Type Definition (DTD) specification, to define the context in which it is valid as a set of rules [Bro96]. An example stick-e note is shown below specifying a short note to be displayed in a particular context.

```
<note>
 <with>Joe</with>
 <at>Dublin University</at>
 <content>Arrange meeting</content>
</note>
```

When the rules evaluate as true based on sensory input, i.e. the context is reached, the note is displayed to the user (in a contextual retrieval or inference process known as *triggering* [PRM99]). The use of SGML eases the process of publishing and exchanging notes, so a repository of notes may be kept on a network server accessible from a user PDA by a wireless link.

3.2.1 Analysis

The stick-e note infrastructure was one of the very first approaches to supporting and simplifying the development of context-aware applications. As such, it succeeds in de-skilling the

creation of context-aware applications, although the infrastructure is designed predominantly to control the display of textual content to the user. The infrastructure provides *SeSensor* components for the abstraction of sensor data, converting sensor readings to SGML fragments. Basic multi-modal sensor fusion is achieved through conjunction of sensor outputs that must be true in order for a context to be active, but no approach is provided to manage the uncertainty of sensor readings. Context data is stored as SGML fragments received from sensors, but no further interpretation is made on the data supplied by sensors to derive higher-level information. The infrastructure does not explicitly provide an inference engine and associated knowledge base, but determination of context is performed by the *SeTrigger* component, which provides a simple inference function. The stick-e note infrastructure provides a simple actuator abstraction in the form of the *SeShow* component that can send commands to any existing program, e.g., to display a note on the screen, and actuation is confined to the presentation of information to a user. Stick-e notes are programmed by creating SGML documents that provide a fairly high-level programming model, but still require significant skills to use. Although the exemplar applications were deployed in mobile environments, the architecture does not provide support for ad hoc mobility.

3.3 Mobile Computing in Fieldwork Environments (MCFE)

The Mobile Computing in Fieldwork Environments (MCFE) project [PMR98] at the University of Kent at Canterbury identifies four generic capabilities required of context-aware applications, namely, sensing, adaptation, resource discovery, and augmentation [Pas98]. Sensing of context and adaptation of application behaviour to context are capabilities shared by all context-aware applications and indeed may be considered base capabilities that make an application context-aware. Resource discovery refers to the capability of an application to discover and exploit other resources relevant in a particular context and is core to the vision of ubiquitous computing [Wei91] where devices dispersed in the environment interact with each other. Contextual augmentation refers to the association of digital data with particular contexts [Pas98], and arises from the nature of the project which is primarily concerned with user interfaces in a fieldwork environment.

The FieldNote system [RJD99] was developed as part of the MCFE project, as a tool for mobile fieldworkers observing wildlife behaviour in Kenya [PRM98, PRB98] and is based on the stick-e note architecture [BBC97], also developed at the University of Kent. The system runs on a handheld computer and augments data collection in the field with location and time information determined using a GPS receiver as a sensor. The system uses a novel XML-based protocol, known as the Context Markup Language (ConteXtML) [Rya99] for exchanging contextual information and field notes between mobile handheld clients and a server. The language also allows context enriched field note data to be queried at the server using spatial and temporal constraints.

The project acknowledges the difficulties of incorporating context information into applications in a generic manner and proposes the Contextual Information Service (CIS) [Pas98] as a dynamic model of contextual information that provides a common access point to the current context for any application. The goals of the CIS are to gather, model, and provide contextual information [PRM99] and the concept is described in some detail as an object-oriented model, but no implementation description or evaluation of the service is available.

3.3.1 Analysis

The MCFE project extends from the early attempts of the stick-e note infrastructure to provide generic support for the integration of context into applications. The context data supported in the project is predominantly geographical location and allied GPS data, with no generic sensor abstraction for the easy incorporation of other sensors. Due to the reliance on a single type of sensor, the project does not provide any treatment of multi-modal sensor fusion. In fact, mono-modal sensor fusion to reduce the uncertainty of individual sensor readings is not dealt with either.

Perhaps the most important contribution of the project is in its approach to the representation of context data by way of ConteXtML. Context information gathered from GPS sensors is represented in this format and stored in persistent storage where it may be accessed and utilised by a range of clients. A similar approach to the stick-e note infrastructure is adopted with respect to inference, although the definition of context-awareness taken by the

project as 'imbuing a device with the capability to *sense* the environment' [PRM98], places less emphasis on inference and the simple triggering approach suffices. Actuation is once again confined to presentation of information to a user and no generic actuator abstractions are defined. The project is specifically focused towards mobile devices, but predominantly supports off-line operation with periodic on-line synchronisation and provides no support for ad hoc mobility. The MCFE project does not attempt to provide a high level programming model and in this respect fails to be accessible to a range of developers. Application development requires specialised skills.

The Context Information Service (CIS) model for the integration of context into applications, although promising, appears to have remained conceptual and does not appear to have been implemented or evaluated.

3.4 SPIRIT project (Bat Ultrasonic Location System)

Research into context-aware computing in Cambridge started at Olivetti Research Ltd. (ORL) between 1989 and 1992 with the design and implementation of the *Active Badge* indoor location system [WHFG92] which used small infrared transmitters (badges) that periodically broadcast their unique identifiers. The broadcast signals are picked up by a network of sensors around a building and by determining which badge was seen by which sensor, the location of the badge and its owner may be inferred. Whilst Active Badge technology allowed location to be determined at the granularity of a room, more interesting context-aware applications required finer-grained location information.

A new indoor location system was subsequently developed by the University of Cambridge Computer Laboratory and ORL that is based on measurement of time-of-flight of sound pulses from an ultrasonic transmitter [WJH97]. The system, known as the *Bat* system [HHS+99] uses a set of small badges (Bats), containing a radio transceiver and ultrasonic transducer. Ultrasound sensors are networked at known locations around a room and monitor incoming ultrasound signals. These signals are transmitted by the Bats when they periodically receive a radio message from a base station. The system is able to determine the 3D location of Bats using a multilateration algorithm, with an accuracy of approximately 3cm [ACH+01].

Coarse orientation information may also be determined.

The Bat location system was used by AT&T Laboratories in Cambridge, as the basis for implementing a *sentient* computing system. The term sentient computing is used in this context to define an application that appears to share the user's perception of the environment and reconfigures itself appropriately [ACH+01]. The sentient system maintains a detailed model of the world containing information about real world entities, modelled using object-oriented techniques. This world model is updated through location information received from Bats and may be used by applications to provide location-aware services such as follow-me systems or a 3D visualisation of the current state of the environment.

The Bat system also uses a form of context-aware quality-of-service adaptation in the location update process. The base stations preferentially allocate location update resources to those Bats that are changing their location (moving) frequently. This is achieved through a scheduling algorithm in the base station, which determines when a Bat will next be addressed to determine its location [ACH+01]. Those Bats moving infrequently may go into a powered down mode to prolong the lifetime of their battery, which in itself is a form of context-aware behaviour.

3.4.1 Analysis

The use of ultrasonic location technologies in the Bat system has enabled very precise indoor positioning and the development of a number of context-aware applications based on location. In addition, a number of *resource monitors* [HHS+99], in effect software sensors, have been installed on networked machines to allow measurement of parameters such as machine activity, resource utilisation, and bandwidth and latency. The system provides some level of abstraction of sensor data, such as conversion of absolute spatial facts to relative spatial facts, although it is not clear how extensible the system is and whether new types of sensor may be incorporated. A systematic approach to sensor fusion is not provided in the system, although sensor data from the ultrasonic badges is filtered before use. Context is represented in the form of a centralised, detailed object-oriented model of the location of entities in the environment and their possible interactions. This abstract model is then made available to

applications that can base their behaviour on the current state of the environment. Inference based on context data is not addressed in the Bats system, but is left to higher-level applications using the world model. A programming model based on a spatial monitoring system is described that allows developers to create new applications based on an event-driven communication style, but it is not clear how accessible and usable this model is to inexpert developers. The Bats system is based on the inherent mobility of location sensors, rather than ad hoc application mobility and is not fully distributed, retaining a number of centralised components.

3.5 The Context Toolkit

The Context Toolkit [SDA99] is an architecture developed at the Georgia Institute of Technology that aims to provide reusable solutions to the problems of developing context-aware applications. The main aim of the toolkit is to free developers from having to deal with the low-level details of context acquisition and allow them to concentrate on the specification of higher-level application behaviours. The toolkit is inspired by the success of toolkits for Graphical User Interface (GUI) development and is based on the GUI concept of a *widget* as a reusable component for abstracting away from and hiding the specifics of a physical device. Through the widget abstraction, the Context Toolkit aims to enable context data to be handled in the same way user input is currently handled [DSFA99].

The architecture was designed using an object-oriented approach and is an implementation of a conceptual framework, illustrated in Figure 3.1, that is composed of five categories of components:

1. **Context widgets** are software components that provide applications with access to context data from their operating environment [SDA99]. Context widgets hide the complexities of individual sensors and abstract raw sensor data to suit the needs of applications, in a reusable manner. Widgets provide callbacks to notify applications of changes, and provide attributes that may be polled by applications [DSA99].

2. **Interpreters** are components that, when provided with state information, can interpret

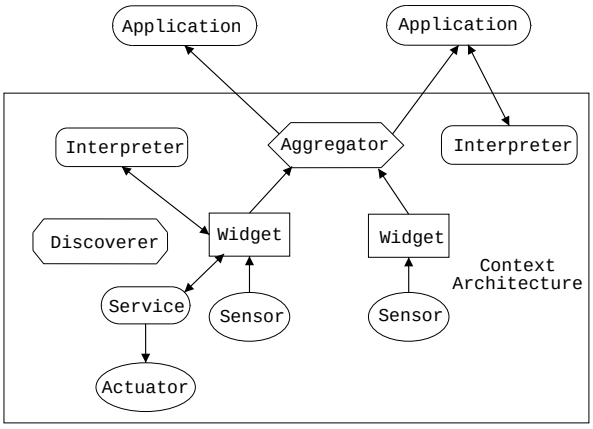

Fig. 3.1: Context Toolkit components and their relationships

the information into another format or meaning [DSA99]. Interpreters transform sensor data to a higher level of abstraction.

3. **Aggregators** provide an additional layer of abstraction between the acquisition of context data and its use, by collecting multiple pieces of logically related context data from distributed sensors and making it available to applications. Aggregators collect multiple pieces of logically related context information into a common repository.

4. **Services** are components within the framework that execute actions on behalf of applications [DSA01]. Services encapsulate actuators in the same way that widgets encapsulate sensors, and provide analogous advantages.

5. **Discoverers** are registries that maintain the set of widgets, interpreters, aggregators and services currently available for use by applications. In the Context Toolkit architecture, this is a single, centralized component.

The toolkit has been implemented in Java and utilises a common communications mechanism based on XML messages over HTTP, making TCP/IP the minimum requirement to support use of the toolkit. A number of applications have been built to demonstrate the usefulness of the toolkit, including an In/Out board, an augmented electronic whiteboard and a conference assistant [DFSA99].

3.5.1 Analysis

The Context Toolkit provides useful domain-specific abstractions for the incorporation of context data garnered from sensors into applications, through the use of the widget abstraction, and this is its major strength. In addition, the interpreter abstraction of the toolkit provides a means to convert sensor data to higher-level context. The toolkit does not deal with issues of uncertainty in sensor data and in fact makes the assumption that the context being sensed is 100% accurate [DSA01], an assumption that is not valid in the real world. As such, the toolkit does not provide a systematic approach managing uncertainty through mono-modal sensor fusion of redundant senor readings. Multi-modal sensor fusion is performed to some extent by the *aggregator* component, although once again, management of uncertainty is not incorporated, with the fusion process assuming complete accuracy of all fragments of data. More recent work on the toolkit acknowledges the uncertainty inherent in sensor data and proposes mechanisms based on components known as *mediators* to reduce ambiguity in sensor data [DMAC02]. However the process is not autonomous and depends on feedback from a user to resolve ambiguous or conflicting sensor data.

The toolkit does not deal with the representation of context data, merely serving the data to higher level applications. Inference and reaction based on context is also left to the application, and the toolkit does not provide generic actuator abstractions to support actuation. The toolkit supports the experienced application developer in integrating context data into applications but does not provide a high-level programming model. As such, the current toolkit is only accessible to experienced programmers, although a proposal has been made to provide a visual environment to support end-user prototyping [DS03].

The context toolkit is designed for use in mobile, distributed environments, however the use of HTTP as a protocol for transmitting events does not address efficient context dissemination [dIn02], and the toolkit does not address ad hoc mobility at all.

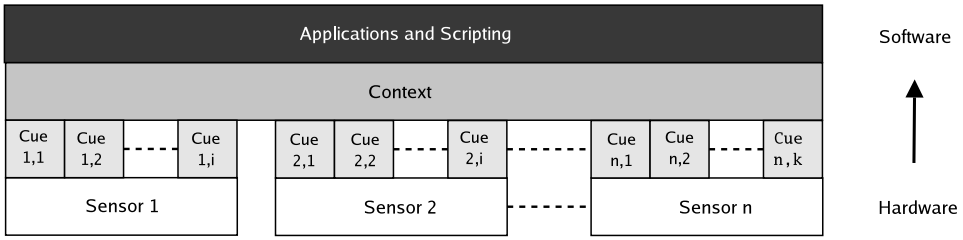

Fig. 3.2: Layered architecture of TEA system [SAT+99]

3.6 Technology for Enabling Awareness (TEA)

The Technology for Enabling Awareness (TEA) project [Lae00] carried out by a consortium of European research institutes aimed to produce a context awareness-enabling add on component for mobile computing and communications devices [LA01]. The project is notable for the fact that it dealt with a range of small sensors measuring a multitude of environmental parameters other than location. A custom hardware board, known as an *awareness device* and incorporating light, sound, motion, temperature, pressure and other sensors was developed [SB98], reflecting the applied nature of the project. The aim of the project was to use this awareness device to provide context data that could be used to adapt the user interface of ultra-mobile devices such as GSM phones and PDAs [SAT+99]. Another application discussed in the realm of the project is the use of context data in communication, enhancing the process of mobile telephone call set-up with situational awareness [STM00].

The awareness device fabricated for the TEA project uses 8 multi-modal sensor devices and a layered architecture is proposed for multi-sensory context awareness and the extraction of context data from sensory data. The TEA architecture is illustrated in Figure 3.2 and consists of the following 4 layers:

1. **The Sensor layer** is the sensor device layer. Two types of sensors are defined - *physical* sensors are hardware components measuring environmental parameters, whilst *logical* sensors measure host device parameters [SB98].

2. **The Cue layer** is a sensor abstraction layer. Each cue represents a single sensor and

different cues may use the same sensor. Cues serve to buffer sensor data and make specific sensor hardware transparent to higher layers.

3. **The Context layer** contains a set of *contexts* as abstract descriptions of a set of situations in which the device could be. A context is based on a number of cues [SF99] and it is this derivation of a single context from a number of cues that can be considered the actual sensor fusion function.

4. **The Applications and Scripting layer** provides scripting primitives to allow developers to incorporate context information into applications.

The way in which transitions are made between the sensor, cue, and context layers in the architecture is of particular interest, since this is how raw data is transformed to meaningful context information.

- Sensor to cue mapping Raw sensor data is mapped to cues through a pre-processing step that serves to extract features that characterise the data over the last period of time [SAT+99]. Statistical functions including average, standard deviation, and quartile distance are used to perform this function.

- Cue to context mapping The architecture proposes two approaches to map a set of cues to a pre-defined context [SB98] (1) explicit rule specifications, enhanced by prior statistical analysis, may be used to infer context from cues; and (2) artificial intelligence methods may be used to relate cues to specific contexts in a learning process.

An example of an explicit rule specification might be 'if cue A has value x and cue B has value y, then the current context is z', while neural networks were used as the basis for the second approach to mapping cues to contexts [CSG99].

3.6.1 Analysis

The major contribution of the TEA project is an architecture for multi-modal sensor fusion in context-aware computing. The architecture provides for both a rule-based and neural

network-based approach to fusing multi-modal sensor data and experiments based on user-interface adaptation on a GSM phone proved the validity of these approaches. The overall architecture for context awareness envisioned by the project represents context information as a set of logical facts augmented with location and time information in the form: Fact=(location, time, value, description) [GBS00]. Another interesting characteristic of the TEA architecture is that it maintains a set of discrete *contexts* in which an entity may exist at a point in time, and recognises that within a specific context, only a subset of all available sensory input is relevant.

The architecture also provides abstractions for dealing with physical sensor devices, but does not provide abstractions for actuation, deferring this to the application level. A high-level programming model based on a set of scripting primitives makes the architecture accessible to even inexperienced developers. The architecture does not however provide a programming model for specifying the sensor fusion implementation (e.g., the rule-base or neural network) for different sets of sensors and as a result the fusion algorithms are restricted to TEA-fabricated hardware devices. Although the architecture is predicated on what the authors term *ultramobile* environments, it does not explicitly provide communication abstractions for dealing with ad hoc mobility.

3.7 Multi-Use Sensor Environments (MUSE)

The Multi-Use Sensor Environment (MUSE) project at the University of California at Los Angeles [CM00, CCKM01] aimed to produce a middleware architecture for smart spaces, i.e., environments pervaded by embedded sensor devices. 'Environment' within the project is not limited to indoor built environments in contrast to much smart space research, but encompasses any environment where the acquisition of information may be important. The project concentrates of the development of new types of *sensing services* and in particular how these services are specified, how QoS is characterised in the services, and how the services can be implemented in terms of resource constraints and performance goals [CM00]. The stated goals of the MUSE project are (1) to develop APIs for the specification of sensing services and the derivation of non-deterministic contextual information from sensor data in

a probabilistic manner; (2) to develop a memory component for context data which permits non-deterministic queries to be made; (3) to optimize device usage based on the quality of derived contextual information whilst staying within the resource constraints of the device [Mun02].

The MUSE infrastructure consists of three major services, namely a lookup service, sensor services, and fusion services. The lookup service is provided by Sun Microsystems' Jini[1], since MUSE service communities are implemented as Jini communities. Sensor services are essentially proxies for sensor hardware that allow the sensor to participate in a Jini federation, and in this way serve to abstract from the complexities of low level sensing. Fusion services are services that derive higher-level contextual information from the data supplied by the sensor services, and it is in this area that the MUSE project has focused, providing *generic* models for fusing sensor data into contextual information.

The MUSE infrastructure models sensing services using evidential reasoning techniques. In other words, sensors are used to supply evidence about the value of a query variable and this evidence is used to determine the most likely value for the variable [CM00]. Bayesian networks are adopted as a basis for interpreting the value of a query variable given sensory evidence, by performing probabilistic sensor fusion. Within MUSE a fusion service is specified by a Bayesian network, with the root node representing the query variable and leaf nodes representing sensors that contribute to the determination of the value of the query variable. Intermediate nodes represent intervening variables. The general structure of a fusion service Bayesian network is illustrated in Figure 3.3. Using Bayesian theory, it is possible to *instantiate* the sensor nodes based on sensor observations, and to observe the effect these observations have on the probability distribution of the query variable.

3.7.1 Analysis

The MUSE project provides a valuable contribution to the provision of generic, multi-modal sensor fusion services for context-aware applications, and the fusion service has successfully been applied to the inference of the location of a wireless client from signal quality measures

[1] http://www.sun.com/software/jini/

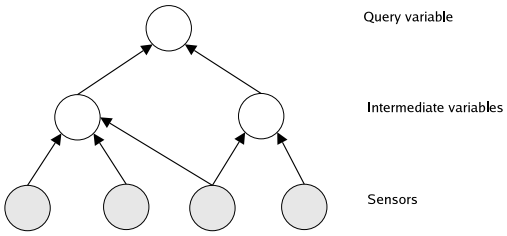

Fig. 3.3: MUSE fusion service Bayesian network

[CCKM01]. MUSE defines sensor abstractions in the form of sensor services which act as smart proxies, independently providing sensor data to interested applications. A subproject, the Multimedia Internet Recorder and Archive (MIRA) project [CMMM00, CM99] deals with the storage of historical contextual data in a 'context database', but the project does not describe a systematic approach to context representation. The project does not incorporate intelligent inference mechanisms, nor does it deal with actuation based on context data, rather simply supplying context data fused from sensor services, to applications. There is no programming model made available to developers to easily specify fusion networks and it appears that whilst MUSE provides a systematic approach to sensor fusion, the advantages are only available to experienced developers. The project uses existing infrastructure in the form of Jini to provide ad hoc service composability in static networks. As such, MUSE does not support ad hoc mobility.

3.8 Context Based Reasoning (CxBR)

Context-Based Reasoning (CxBR) was developed at the University of Central Florida [GA94, GA95] as a concise but rich representation paradigm that could be used to model the intelligent behaviour of simulated entities. The paradigm derives its name from the idea that the actions taken by an entity are highly dependent on the entity's current context. CxBR is based on the following hypotheses:

1. Small, but important portions of all available environmental inputs are used to recognise and treat the key features of a situation. A driver, pilot, or air traffic controller receives

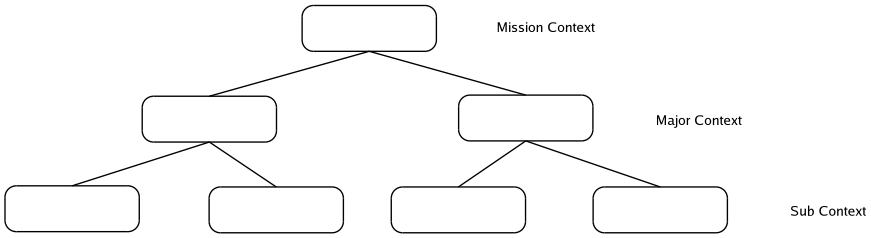

Fig. 3.4: A CxBR context hierarchy

a multitude of audio, tactile and visual inputs whilst performing his job. These inputs are all easily handled whilst in the normal range, however an input deviating from the normal will cause the operator to focus only on that input in order to recognise the situation.

2. There are a limited number of things that can realistically take place in any situation. This fact may be used to narrow the number of potential actions in a situation and speed up response to a situation.

3. The presence of a new situation will generally require alteration of the present course of action to some degree.

CxBR is based on military tactics where a specific situation demands a strict, pre-defined set of actions be embarked upon and the problem becomes two-fold (1) recognition of the present situation; and (2) determination of action to be undertaken when the situation is recognised [GA99]. These problems are managed by defining the set of situations in which an entity may be during its lifetime, and within each situation defining the set of other situations to which a transition is possible (since a particular situation inherently limits what other situations may follow). Situations are represented as *contexts*, each context being mapped to a class encapsulating behaviour as a set of functions.

The CxBR approach defines a *mission context*, *major contexts* and *sub contexts* structured in a three-level hierarchy as illustrated in Figure 3.4. The mission context represents the overall goal and objectives for a certain scenario. It is composed of major contexts, which

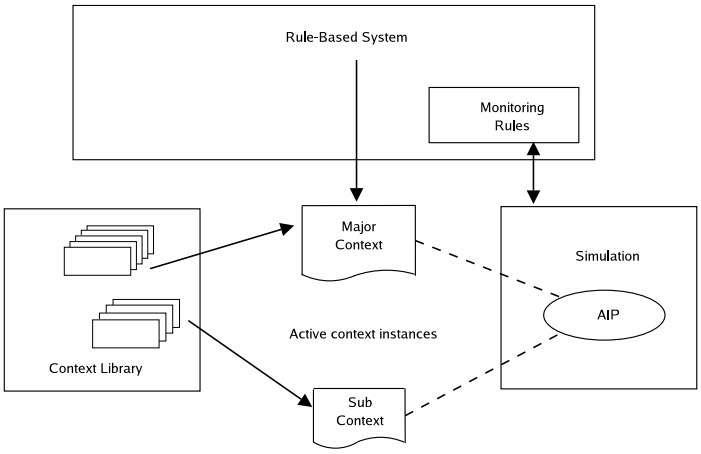

Fig. 3.5: CxBR system diagram [GA99]

are tactical operations assisting in the achievement of the scenario. Each major context is in turn composed of one or more sub contexts each of which is a lower level tactical procedure that assists in the achievement of its associated major context. Sub contexts are of a finite, and typically short, duration.

Autonomous behaviour of an entity in CxBR is based upon the evaluation of a set of production rules, and the alteration of behaviour based on the outcome of these rules. The influence on behaviour based on a set of continuously evaluated rules is a common Artificial Intelligence technique, but CxBR adds to this the notion of an *active context*. Within each active context, only a subset of all rules available in the system are evaluated (this derives from the hypothesis that there are only a limited number of things that can realistically take place in any situation) and this increases the efficiency of the inference process. Additionally, within each context, only a subset of all available sensor inputs need to be monitored, further reducing complexity. Contexts are constantly being activated and deactivated during an entity's lifetime and each context regulates the behaviour of the entity and provides an expectation for the future. A requirement of this approach is that certain cues exist to indicate when transitions may be made between active contexts.

The structure of the CxBR system is illustrated in Figure 3.5 and illustrates how a major context is the main control element and determines which rules are currently active.

Monitoring rules identify when a transition to another major context is indicated by the situation, and in the case of a sub context, check for completion of the sub context action(s) [GA95]. The mission context simply serves to define application parameters and define the set of required major contexts.

3.8.1 Analysis

CxBR provides a framework to simply, concisely, and efficiently represent and reason about context data derived from sensors. In a traditional rule-based system, the task of situational assessment is of complexity $O(n)$ where n is the total number of rules in the system. CxBR reduces this complexity to O(k) where k is the average number of rules attached to active contexts and significantly, k is independent of the total number of rules in the system [GA99]. The system has been tested in a vehicle simulation [GPG00] with promising results in the efficiency of the approach. CxBR does not provide sensor or actuator abstractions at any level, nor does it address uncertainty of sensor data and its management using sensor fusion techniques. No programming model or support is offered for application development, rather the salient contribution of the system is the provision of an efficient and systematic approach to context representation and inference.

3.9 GUIDE

The GUIDE project at the University of Lancaster developed a context-sensitive guide for tourists visiting the city of Lancaster [DMCF99]. The system is based around mobile hand-held tablet PCs communicating with a GUIDE server via base stations located within cells of high bandwidth wireless communication covering selected areas of the city. The premise behind the project is that tourists equipped with the tablets wander the city and are provided with information tailored to both their location and personal preferences. In contrast to earlier, similar, systems, e.g., [LKAA96], GUIDE adopts a distributed networked approach with information being dynamically composed at the client from both local and remote fragments of information [CMD99], which makes the system more dynamic and flexible. Location in-

formation is determined from the current cell, so location and information are disseminated on the same channel. Information is broadcast by cell base stations to minimise the effect on response times of having more than one unit present in a cell [CDMF99].

3.9.1 Analysis

Whilst the provision of generic support for the development of context-aware applications is not one of the explicit goals of GUIDE, the GUIDE project provides valuable insight into one of the few successful context-aware systems in use by the general public and the lessons learned [CDM+00] in the development of such an application.

The GUIDE system does not handle multi-modal sensory information (rather relying solely on location information from strategically placed beacons) and does not deal with sensor abstraction or multi-modal sensor fusion. The experiences of GUIDE have been further generalised to strategies for building context-aware applications [CDME00], although the focus is still on interaction with a human user. Two major issues in engineering context-aware systems which arose from the project are (1) the need for both *accurate* and *timely* sensor data; and (2) the need for flexibility in order to adapt to problems inherent in determining context. The major issue raised of the accuracy and timeliness of sensor data reinforces the requirement for an efficient approach to sensor fusion to reduce the uncertainty of context data. The GUIDE project successfully uses an object-oriented approach to represent context data, most importantly location, but also identity and certain user preferences [CDM+00]. Simple inferences are made on the context data, with the results being used to alter information presented to a user. With actuation being limited to the presentation of information, no actuator abstraction is defined within GUIDE. The system is reliant on installed infrastructural network access points and does not cater for ad hoc communication.

3.10 Context Fabric

The Context Fabric project being carried out by the Group for User Interface Research at the University of California at Berkeley [HL01] proposes a novel approach to providing support

for context-awareness in the form of a *service infrastructure* model. This model attempts to shift as much of the task of context-aware computing as possible to a network-accessible middleware. This approach aims to aid the development of applications based on a diverse and constantly changing set of sensors and devices by providing uniform abstractions and reliable services for common operations. The Context Fabric provides five basic *context services* as an integral part of the infrastructure. The relation between these constituent services is illustrated in Figure 3.6 and they are described below:

1. **Automatic Path Creation** is a service which simplifies the task of refining and transforming raw sensor data into higher-level context data.

2. **Proximity-Based Discovery** is a service which provides a device with information about all sensors near to it.

3. The **Context Event Service** provides an event system to asynchronously communicate context data using the Context Specification Language (CSL) that provides a declarative approach to stating context needs at a high-level.

4. The **Context Query Service** provides a universal interface by which applications can synchronously check context state.

5. The **Sensor Management Service** provides a registry of all local sensors and deals with discovery and registration of sensors.

The fabric services are held together with a Context Specification Language [Hon02] which is a simple XML-based language that supports context event management and provides a declarative approach to state context needs at a high level (e.g., what bank is closest to me?). The language purports to support event subscriptions filtered on proximity (e.g., subscribe to events near me) as part of the implementation of the proximity-based discovery service, but it is not clear how this notion of proximity is implemented in the system. In addition, the CSL also supports context queries, whereby applications can query the infrastructure for specific context data.

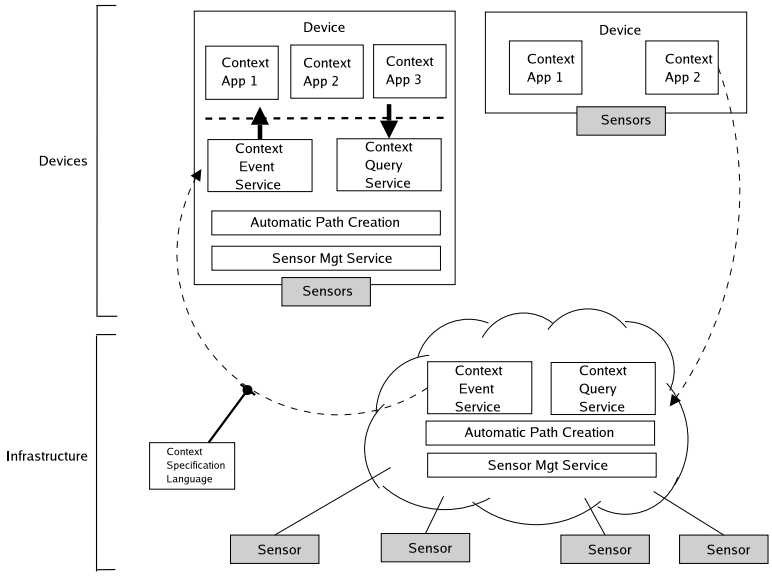

Fig. 3.6: Architecture of the context fabric [Hon00]

The advantages of the infrastructure approach to context-aware computing are argued to be (1) an independence from hardware, operating system and programming language; (2) improved maintenance and evolutionary capabilities; and (3) sharing of sensors, processing power, data and services.

The context fabric stores context data in network-addressable, logical units known as *infospaces* [HL04], which contain both static and dynamic data. Sensors change infospace data, and context-aware applications can examine infospaces and retrieve the data stored therein. Within an infospace, context data is stored as a set of typed *tuples*. Tuples are represented as XML documents.

3.10.1 Analysis

The Context Fabric provides sensor abstraction through the APC service and is one of the few projects that provides an explicit treatment of proximity as a useful concept in context-aware computing. Sensor fusion as an approach to managing the uncertainty of sensor data is not dealt with in the project, although logically it would take place in the APC service. Context

data is represented as a set of tuples expressed in XML, and against which queries may be made. As the project is concerned with simply providing context data to higher level applications it does not provide an approach to intelligent inference, although the proximity-based discovery service uses a form of inference in determining proximity. Furthermore, actuation is not dealt with and is left to individual applications. The project provides a programming model based on infospaces which allows developers to incorporate context information into applications based on a hybrid blackboard and dataflow architecture [HL04]. Furthermore, the CSL provides a structured query language through which to access context data. Although the potential applications of the Context Fabric are inherently mobile, the project does not explicitly provide support for ad hoc mobility.

A potential disadvantage of the infrastructure approach is the potential for a single point of failure within the system, and the Context Fabric attempts to mitigate this risk by storing context data in multiple distributed locations.

3.11 Target Recognition using Image Processing (TRIP)

The Target Recognition using Image Processing (TRIP) project is the result of Ph.D. research [dIn99, dIn02] at Cambridge carried out in conjunction with AT&T Laboratories. TRIP uses a novel vision-based sensor technology to determine location and identity information that allows applications to create a model of their environment on which they may base their behaviour in a *sentient*[2] manner [dInL01]. The project develops an inexpensive indoor location sensor which uses low cost CCD cameras and 2-D *ringcodes* (known as *TRIPtags*) [dIn01] to imbue applications with location awareness. TRIPtags can be printed and attached to objects which may then be located relative to cameras dispersed in the environment, using algorithms to determine the exact location of the TRIPtag with respect to the camera [dInMH02].

A number of components were developed within the project in order to ease the development of location-aware applications.

[2]The term *sentient computing* is used as an analogy for *context-aware* computing in the TRIP project

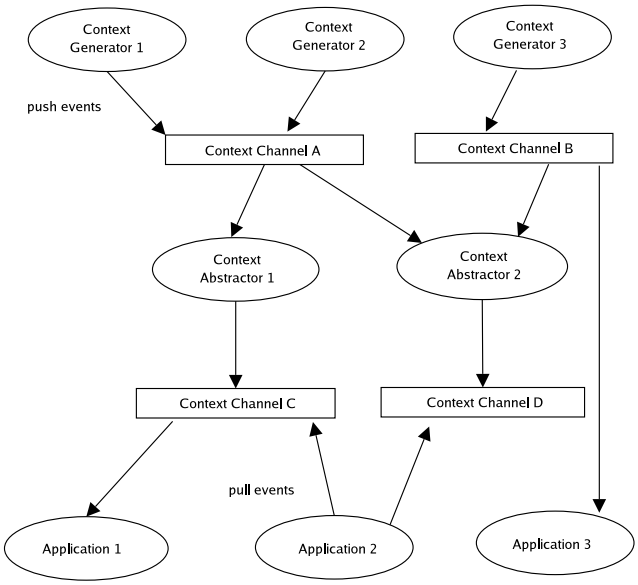

Fig. 3.7: The SIF architecture [dIn00]

3.11.1 The Sentient Information Framework

The Sentient Information Framework (SIF) is a set of co-operating distributed software components that use events to transmit context notifications, and was developed to isolate context capture from application semantics [dIn00]. The framework consists of 3 components and is illustrated in Figure 3.7.

1. **Context Generators**

   A Context Generator encapsulates one or several sensors and serves to simply transfer raw sensor data to the Context Abstractor, as a series of events. This component does not abstract the raw sensor data in any way.

2. **Context Abstractors**

   A Context Abstractor accepts raw sensor data in event form from a Context Generator, and enhances the data through interpretation and augmentation from a database. Interpretation of data is performed by applying conditional rules, which if they succeed, generate an action. Such actions cause enhanced contextual events to be passed onto

the application. An event can flow between any number of abstractors before delivery to an application.

3. **Context Channels**

   A Context Channel provides for communication between other SIF components and is defined by the type of events it carries. Context Channels serve to de-couple the interaction between SIF components and applications by allowing multiple producers and multiple consumers to communicate with each other transparently and asynchronously. The channels are implemented as OMG Notification Channels [dIn01].

Parallels are evident between the SIF and the Context Toolkit (see section 3.5) and indeed the SIF was inspired by the toolkit amongst other work [dIn00]. The SIF differs most significantly from the toolkit in its use of a CORBA-based event service which serves to de-couple the components, which is in contrast to the tight coupling of components in the Context Toolkit.

3.11.2 Event-Condition-Action (ECA) Rule Matching Service

Another component of TRIP addresses the limitations of the CORBA Notification Service in only being able to filter on atomic events. In sentient applications, reactions are usually based on a combination of atomic events, and the Event-Condition-Action (ECA) rule matching service is a middleware service that undertakes the common event composition and aggregation tasks required in the implementation of reactive systems [dInK01]. The existing capabilities of a production system language, CLIPS [NAS99] are leveraged to associate aggregated events to actions.

The ECA server architecture is illustrated in Figure 3.8 and shows the different components of the architecture. The Rule Registration module converts rules issued in a custom rule specification language into CLIPS rules and passes them to the inference engine. The Event Reception module takes events from distributed event sources and maps them to CLIPS facts that are understood by the inference engine. The Notification Dispatcher takes CLIPS facts representing aggregated events and maps these onto a batch of CORBA events that are

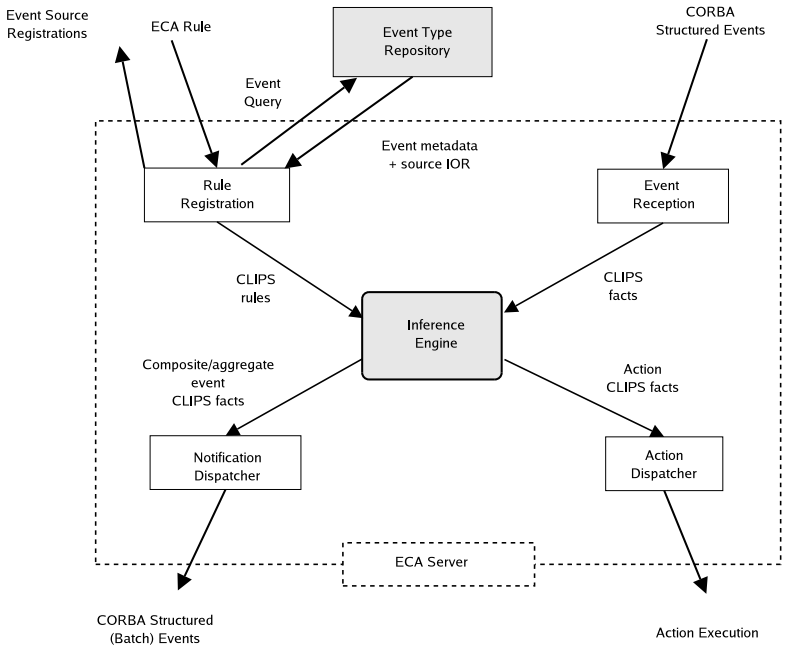

Fig. 3.8: The ECA Server system [dInK01]

notified to clients. The Action Dispatcher module processes facts from the inference engine and triggers actions based on these facts.

Another feature of the ECA-Rule matching Service is the provision of a domain-specific language in the form of the ECA rule specification language, the function of which is to specify the set of patterns to be applied to the contents of events together with the actions that are triggered by successful matches [dInK01]. The function of the language is to enable the creation of sophisticated rules expressing complex conditions upon atomic events and remove the burden of composite event handling from the programmer.

3.11.3 Analysis

TRIP makes significant advances in the provision of support for the development of location-aware systems. Although implementing only one kind of location/identity sensor, the system specifies a framework for the development of applications using a multitude of sensors. The framework contains a sensor abstraction component and uses event-based communication to

decouple the constituent components. TRIP does not explicitly define a systematic approach to multi-modal sensor fusion, but provides support for it within the Context Abstractor component of the SIF, and through the event composition capabilities of the ECA Rule-Matching service. Context information is represented through the assertion of logical facts in a knowledge base, whilst intelligent inference is achieved through CLIPS rules. Actuation within TRIP is software based, with no abstraction of physical transducer devices. TRIP significantly eases the development of reactive applications with the provision of the ECA rule specification language for programming reactive behaviour, but does not provide a federated, systematic approach to application development encompassing sensor fusion. TRIP provides mechanisms for object lifecycle and location control supporting mobility at the object level, based on the migration and remote instantiation of CORBA objects, but does not provide support for ad hoc mobility.

3.12 Gaia

The Gaia[3] project at the University of Illinois at Urbana-Champaign proposes a general-purpose operating system for ubiquitous computing environments, which exports and coordinates the resources contained in a physical space [CHRC01]. The project aims to make physical spaces containing a plethora of computation and communication devices into programmable systems. These systems are known as *active spaces* and the Gaia OS has been developed as a component based, distributed metaoperating (an operating system under which several other operating systems are active) system to manage the resources contained in an active space [RC00].

The major components of the OS are the Gaia kernel and the application framework [RZC03]. The kernel consists of a Component Management Core for creation, destruction and uploading of components, and five services built as Gaia components, namely event management, the context service, the presence service, the space repository and the context file system [RHC+02]. The application framework sits logically above the kernel and models applications as a set of distributed components, re-using concepts from the Model-View-

[3]After James Lovelock's **Gaia** theory of the earth as a single, self-regulating system

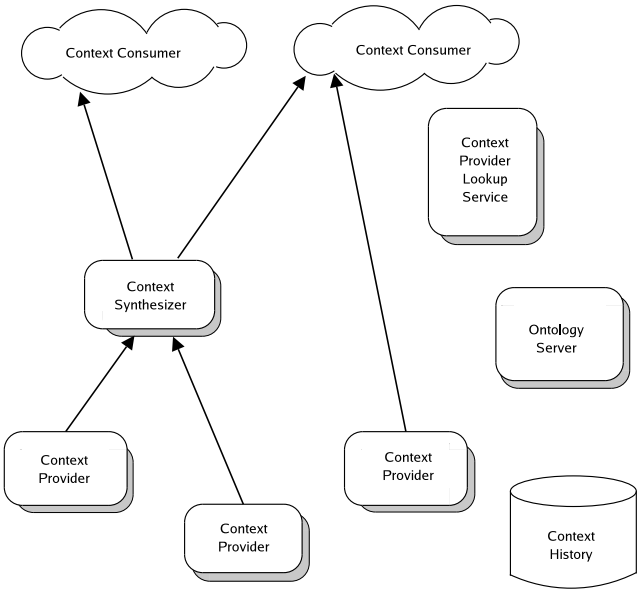

Fig. 3.9: Gaia Context Infrastructure [RC03a]

Controller design pattern. According to [RC02], the framework provides functionality to alter the application composition dynamically, is context-sensitive, supports the creation of active-space independent applications and provides policies to customize aspects of the application including mobility. Active space independent applications are provided for through the use of generic application description files and application description files customized to specific spaces.

3.12.1 Context Model

The context model used by Gaia is based on the use of predicates to describe context information derived from sensor data. Ontologies are employed to describe the structure and properties of context predicates. A context predicate has a name which describes the type of context that the predicate describes, and a set of arguments that may include relational operators. Context predicates are specified using an ontology that defines the set of allowable context types and their legal arguments. As well as allowing the validity of specific context predicates to be checked, the ontology permits semantic inter-operability between different

applications by providing a shared vocabulary.

3.12.2 Context Infrastructure

Gaia defines a *Context Infrastructure* [RC03a], illustrated in Figure 3.9 and consisting of the following components:

- *Context Providers* represent sensors or other providers of context data. Context providers may contain logic and may use different mechanisms to reason about the contexts they sense and to answer queries, e.g, first-order predicate logic [RC03a]. Context providers may both *push* sensor data by publishing events, or it may be *pulled* from them using a Prolog-like query language.

- *Context Synthesizers* serve to deduce higher-level contexts based on inputs from Context Providers and in this way act as context providers themselves. Context synthesizers use either rules to infer higher-level contexts (rule-based synthesizers), or machine learning techniques based on algorithms such as the Naïve Bayes algorithm.

- *Context Consumers* are applications that get context information from providers or synthesizers, reason about it, and adapt their behaviour accordingly. Context consumers can use either a rule-based approach to specifying application behaviour, or learning approaches such as neural networks or reinforcement learning techniques.

- *Context Provider Lookup Service* serves as a registry with which context providers register the context they provide

- *Context History Service* stores past contexts in a database. Gaia also allows event channels to be made persistent and this data to later be mined to determine patterns.

- *Ontology Server* maintains ontologies that provide semantic descriptions of types of contextual information

3.12.3 Analysis

Gaia is designed to facilitate the development of applications in physically bounded environments rich in heterogeneous devices. The system is component based and defines sensor abstractions in the form of context providers and implements asynchronous event-based communication between system components in addition to synchronous polling. An important contribution made by Gaia is its ontological context model. By using an ontology to describe Gaia context predicates, interoperability between pervasive system components is greatly enhanced. However, the use of the DAML+OIL language[4] for describing the ontology is not ideal. DAML+OIL was designed for the Semantic Web and not suited to some aspects of ubiquitous computing and does not easily deal with quantitative concepts such as order, quantity, time, geometry and probability. Nevertheless, as an approach to semantic interoperability the context model is promising. Gaia proposes two possible approaches to sensor fusion, which may be applied at either a context provider or a context synthesiser. Both components offer the ability to fuse fragments of sensor data with providers offering monomodal sensor fusion and synthesisers multi-modal fusion. Both components are able to fuse data using either rules-based or machine learning techniques, although detailed application examples are not offered so it is difficult to evaluate this approach. Furthermore, Gaia makes use of Bayesian networks to reason about uncertain contexts, specifically attaching probabilities derived from Bayesian networks to context predicates [RAMC04]. Gaia uses the Prolog production-rule based system to perform inference on context data, an approach that is closely aligned to the use of logical predicates to represent context data. Gaia provides a programming model based both on a high-level scripting language and graphical tools, aiming to ease the development of active space applications. Mobility in Gaia is defined at the level of an active space, and addresses mobility of application components between devices within an active space, and across different active spaces. As such, loosely coupled communication supporting ad hoc networking is not specified in Gaia.

[4] http://www.daml.org/language/

3.13 Solar

Solar is a software infrastructure developed at Dartmouth College. Solar proposes a graph abstraction for the collection, aggregation and dissemination of context data. The Solar infrastructure is based on an asynchronous event-based communication mechanism and the context-aggregation process is decomposed into a set of modular and re-usable operators [CK02b], in a tree-structured namespace [CK02a]. Operators are objects that subscribe to a particular event stream, process incoming events, and publish another event stream. Operators may be recursively joined to form a directed acyclic graph to collect and aggregate desired context [CK02b].

The operator graph consists of three types of nodes (1) *sources* are wrappers for context sensors and produce events; (2)*operators* subscribe to events from sources and publish an event after processing an input event and; (3) *applications* are event sinks and subscribe to events, to which they react.

An example operator graph is illustrated in Figure 3.10, with sources shown as empty circles, and applications as empty squares. There are four categories of operators defined, depicted as circles with a letter defining the category of operator. A filter outputs a subset of the events it consumes, whilst a transformer outputs a different type of event to that which it consumes. A merger outputs all events it receives, whilst an aggregator outputs an event type based on events in one or more input streams.

Naming in pervasive environments is a challenging proposition with any solution required to be fast and scalable, as well as flexible and adaptable. Solar incorporates a naming system based on the Intentional Naming System (INS) [CK03].

3.13.1 Solar architecture

The Solar system consists of a number of components as illustrated in Figure 3.11. Operators reside on *planets*, which are the execution platform for sources and operators, tracking subscriptions, and delivering events in the operator graph. Sensors (sources) register with a planet to advertise their availability. Applications may connect to planets to select appropriate sensors and aggregated output streams [CLK04]. *Stars* maintain a representation of

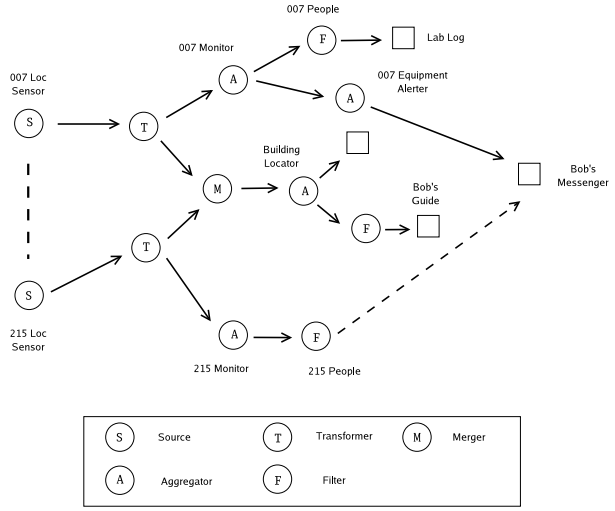

Fig. 3.10: An example Solar operator graph

the operator graph and service requests for new subscriptions [CK02b]. When a star receives a new subscription request, it attempts to find operators that may be re-used, essentially mapping an operator graph to the planetary network in order to distribute the load. The naming service allows stars to resolve name queries on new subscription requests.

3.13.2 Analysis

As an approach to support the development of mobile, context-aware applications, Solar is based on the collection and aggregation of sensor data, and the dissemination of higher level context data by a set of servers in the network rather than by individual applications. Solar is based on an event-based communication abstraction, where changes in context data are communicated asynchronously as events. As such, Solar defines sensor abstractions as event producers, encapsulating both hardware transducers and software sensors. In the Solar model, sensors 'push' events, and cannot be queried by other entities. Solar provides an approach to sensor fusion based on a graphical network of operators, the fusion function being performed once again by servers in the network rather than by applications. Since Solar is predominantly concerned with the *supply* of context data to applications, it does

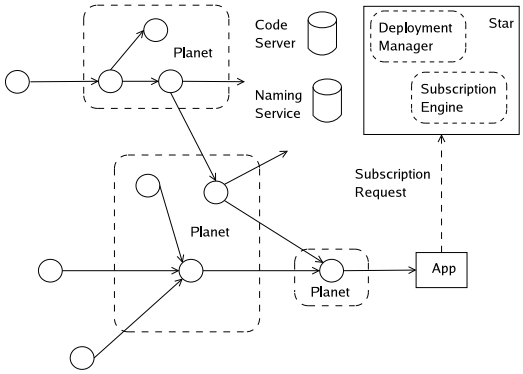

Fig. 3.11: Solar architecture [CK02c]

not deal with the representation of context data. However, some Solar operators maintain state as part of their function, but no systematic approach is provided and the mechanism employed is operator-specific. Due to Solar's emphasis purely on the supply of context data to higher level applications, the architecture does not deal with inference based on context data at all. A programming model is provided allowing the composition of operator graphs, and there is a library of existing operators. In addition, other operators may be developed by extending existing base classes and implementing abstract methods, although there is no detailed treatment of the programming model available. Solar is designed specifically for mobile environments, but no support is offered for ad hoc device mobility.

3.14 Observations

Our review of state-of-the-art approaches to supporting the development of mobile, context-aware applications revealed that there are a range of approaches broadly aimed at easing the development of such applications, making the process more accessible to a greater number of developers. Based on the set of requirements identified in section 2.3.8, we are able to evaluate the extent to which the state-of-the-art supports these requirements.

Requirement	Stick-e notes	MCFE	SPIRIT	Context Toolkit	TEA	MUSE	CxBR	GUIDE	Context Fabric	TRIP	Gaia	Solar
R1: Loosely coupled comm^n												
R2: Sensor abstraction	•		•	•	•	•			•	○	•	•
R3: Sensor fusion	○			○	•	•				○	•	•
R4: Context representation	•	•	•		•		•	•	•	•	•	
R5: Inference engine										•	•	
R6: Actuator abstraction	○											
R7: Developer support	○		○	•	○				○		•	

• = support ○ = limited support

Table 3.1: Features provided by state-of-the-art approaches to developing mobile, context-aware applications

3.14.1 Existing support for requirements

Whilst each of these projects goes some way in providing what we consider the critical requirements of a programming model to support the development of mobile, context-aware applications, there is currently no system that provides support for all requirements. Table 3.1 illustrates what we consider to be the essential set of requirements for which support is required and evaluates each of the projects discussed in this chapter with regard to whether they provide support for the requirements. The requirement for developer support is treated as a unifying requirement, exposing support for other requirements to the application developer in an accessible manner likely to promote pervasive application development. In the table, a bullet (•) indicates support, whilst a circle (○) indicates limited support.

We have identified middleware that provides a higher level abstraction of raw sensor data that it serves to applications without fusing it with other data, or performing any inference on it (e.g., SPIRIT, Stick-e notes, Context Toolkit). Other approaches provide mechanisms to fuse multi-modal sensor data in order to derive higher level contexts, but do not provide an easily accessible programming model to application developers, or any form

of actuator abstraction (e.g., MUSE, TEA). The representation of context information is treated thoroughly in some projects (e.g. CxBR, GUIDE), but these do not deal with sensor fusion, nor provide a programming model. Other projects provide good support for sensor abstraction and sensor fusion (e.g.,TRIP, Gaia, Solar, Context Toolkit) but do not provide suitable communication paradigms for ad hoc mobile networks.

3.15 Summary

In this chapter we introduced a number of state-of-the-art approaches to support the development of context-aware applications and evaluated them against the set of requirements we consider important in an architecture for developing mobile, context-aware applications. We found that whilst individual projects together treated most of the important requirements to some extent, an architecture providing a unified approach encompassing all the features has yet to be proposed. We take all these components into account in the design of a new approach to the development of mobile, context-aware applications.

The next chapter describes the sentient object model which provides systematic support for the development of context-aware applications in mobile, ad hoc environments.

Chapter 4

The Sentient Object Model

The previous chapter examined state-of-the-art approaches to supporting the development of mobile, context-aware applications with respect to the set of requirements outlined at the conclusion of chapter 2, and found no single approach that supported these requirements. This chapter describes our programming model, based on the sentient object model. The design of the model is based on the set of requirements we identified as being necessary to support developers of mobile, context-aware applications. The model fulfills these requirements, providing a systematic approach to the design and development of context-aware applications in mobile environments.

The design of the sentient object model is directed by the nature of the environment in which applications are intended to execute. The major characteristics of the environment influencing the design of the model are the dynamism and limited bandwidth of ad hoc networks, coupled with the need to capture, represent, and process context efficiently in such environments.

4.1 Introduction

The Merriam-Webster dictionary[1] defines *sentient* as 'responsive to or conscious of sense impressions'. The key aspect of this definition to note is that sentience is defined as being

[1] http://www.m-w.com

responsive to sensory input. We are not the first to employ this term in the area of ubiquitous computing, but attach slightly different semantics to the term. Addlesee et al. [ACH+01] use the term sentient computing to describe applications that appear to share the user's perception of the environment and are able to react to changes in the environment according to a user's preferences. Similarly, de Ipiña defines sentient systems as systems that respond to stimuli provided by sensors distributed throughout the environment by triggering actions that are adequate to the changing context of the user [dInK01]. The Economist [Unk03] sees sentient computing as primarily a reactive technology to enhance user interaction with applications.

In contrast to current definitions of sentient computing as primarily *reactive* systems enhanced with environmental inputs, we define sentience as the ability to perceive the state of the environment via sensors, and use this information to *proactively* actuate on the environment. Proactiveness is defined as the ability of the system to act in anticipation of future states or goals of the system. This is in contrast to reactive systems where the output of the system at time t is dependent only on past states. In proactive systems, the output is dependent on future states too. It is this recognition of proactiveness that distinguishes our definition of sentience from previous work, and we use the term to mean proactive context-awareness. Furthermore, our definition of context-awareness is focused on *autonomous* systems, rather than more traditional approaches that define context-awareness as merely enhancing user interaction with a computer system (see chapter 2). This focus is based on our belief that truly pervasive computing should not be exclusively user-interaction-centric.

The sentient object model defines a set of software abstractions that together provide a programming model supporting the developer of context-aware applications in mobile ad hoc environments and significantly eases the development task. In the following sections, we describe the sentient object model in terms of the fulfillment of the seven major requirements identified in section 2.3.8.

4.2 Loosely coupled communication

As discussed in section 1.2, we view interaction between the environment and application components as the basis of context-awareness, which in turn is crucial to the realisation of truly pervasive computing. Support for a suitable communication mechanism that addresses the challenges of the envisioned pervasive computing environment, underlies the first requirement of the programming model.

Mobile ad hoc networking is one of the key enabling technologies for pervasive computing since in a truly pervasive computing environment, information exchange between entities cannot rely on a fixed network infrastructure and ad hoc peer-to-peer wireless connections will be the norm. This is due to the fact that fixed and infrastructure networks are too expensive to deploy ubiquitously, and indeed, mobile devices may not be networked this way.

A variety of communication middleware solutions based on a synchronous style of communication exist for fixed networks, e.g., Java Remote Method Invocation (RMI) [WRW96], the Common Object Request Broker Architecture (CORBA)[Gro02] and the Distributed Component Object Model (DCOM) [HK97]. As discussed in chapter 2, these solutions typically assume constant and reliable connections and are based on synchronous communication between distributed components, with the identity and address of interacting entities known a priori. The assumptions made by such synchronous, connection-oriented middleware do not hold in mobile, resource-constrained environments and in contrast, an anonymous, generative, style of communication is best suited for applications in these environments, where asynchronous notifications of changes in context free applications from the need to resort to expensive polling behaviours. It is based on the characteristics of communication in mobile, ad hoc networks that we derive our first requirement for supporting the development of context-aware applications in such environments, namely the requirement for an anonymous, generative style of communication.

Event-based communication [BMB+00] provides an anonymous, generative communication paradigm suited to the characteristics of mobile environments where communication relationships between dynamically changing populations of loosely-coupled entities change regularly over the lifetime of the application [MC03]. This style of communication abstracts

from details such as the identity and interface of co-operating entities and allows the application developer to focus only on information required and produced by entities [HB98].

A number of event-based communication services have been developed for asynchronous communication in distributed systems. The Scalable Internet Event Notification Architecture (SIENA) is an event notification service providing a scalable, general-purpose event model for highly distributed applications residing in a wide-area network that require component interactions ranging in granularity from fine to coarse [CRW01]. Events in SIENA are defined as sets of typed attributes, against which filters may be applied to determine event delivery. The operational semantics of SIENA are based on advertisements and subscriptions between event producers and consumers. The SIENA architecture is based around a set of intermediate event servers, which is not suited to the ad hoc environment envisioned by the sentient object model due to the lack of infrastructure to host event servers, and the likelihood of network partitions occurring between application components. SIENA does not provide any support for proximity-based filtering of event notifications.

The Java Event Distributed Infrastructure (JEDI) is an object-oriented infrastructure supporting the development and operation of distributed event-based systems [CNF01]. The JEDI architecture is based around *event dispatchers* supporting subscription to, and delivery of, events. Events in JEDI are represented as ordered strings composed of the name of the event, and a set of parameters against which filters may be applied. Mobility is supported in JEDI through the use of mobile agents able to migrate across network nodes and provides support for temporary disconnection of event producers and consumers. The use of intermediate components to propagate events by JEDI is impractical due to the reliance on fixed network infrastructure which is not available in an ad hoc network environment.

Scalable Timed Events and Mobility (STEAM) is an event service designed specifically for mobile, ad hoc networks and is fully distributed over mobile ad hoc nodes within a network. A number of characteristics of STEAM make it the most suitable candidate to address Requirement 1 within the sentient object model, identified in Chapter 2.

4.2.1 Scalable Timed Events and Mobility (STEAM)

Interaction between components in the sentient object model is based on STEAM [MC03], an event service based on an implicit event model and designed specifically for mobile ad hoc networks. STEAM has a number of unique characteristics that distinguish it from other distributed event-based middleware and make it suitable for interaction in pervasive computing environments [Mei03]:

1. Mobility support - the middleware uses ad hoc wireless communication between application components, obviating the need for pre-deployed communication infrastructure.

2. An inherently distributed architecture - the middleware is collocated with application components and does not rely on any centralised components.

3. Location-aware application components - geographical location information is provided to individual application components by a location service.

4. Distributed event notification filtering - filtering of event notifications is available both at the producer and the consumer using different filtering mechanisms.

5. Geographical scoping of event propagation - location awareness permits the validity of event notifications to be scoped to a specific geographic area.

STEAM is designed for highly mobile, collaborative applications, that is those applications that work jointly with others, and addresses the requirements of such applications through proximity-based event notification dissemination. This use of proximity to bound dissemination is unique in event-based middleware, and allows physically mobile application components to communicate when physically close together.

The event service has an inherently distributed architecture whereby the middleware is exclusively collocated with application components and there are no centralised components, as illustrated in Figure 4.1. This means STEAM can operate over ad hoc networks, with no need for a fixed infrastructure hosting centralised components.

STEAM employs an *implicit* event model [MC02b], whereby producers publish events of specific *types*, whilst consumers subscribe to events of particular types, rather than producing

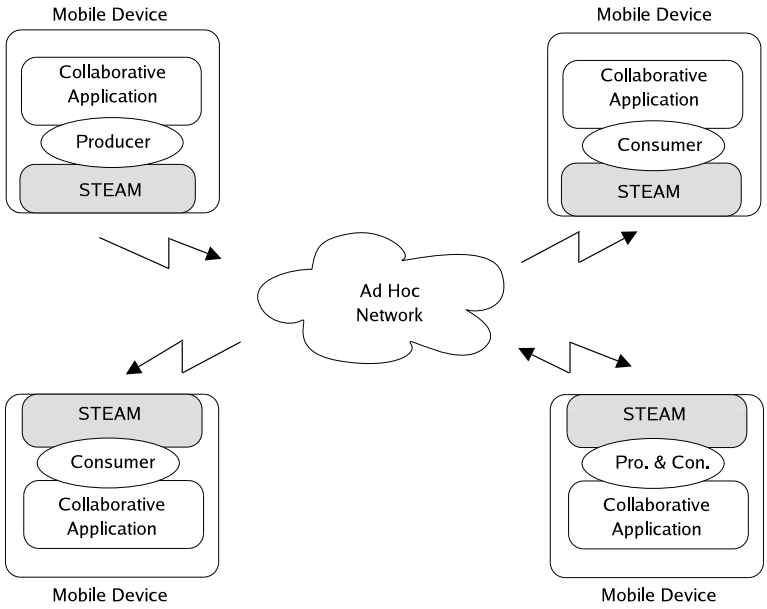

Fig. 4.1: STEAM event model (adapted from [Mei03])

entities or intermediate entities. Prior to publishing event notifications, producers *announce* the type of event(s) they will publish. Each event type announcement has an associated geographical proximity in which the event type will be published, and only subscribed consumers within this proximity will receive event notifications.

Three different types of filters are supported by STEAM that may be applied to event notifications. A combination of the three types of filter may be applied and an event notification is only delivered if all filters match.

1. **Subject** filters match on the type or *subject* of the event and are specified at the *producer* and *consumer* (implicitly through announcements and subscriptions) and events are only propagated if there exists a consumer interested in the event type.

2. **Content** filters match on the values of the parameters of a specific event instance and are specified at the *consumer* and are evaluated on reception of an event at a consumer.

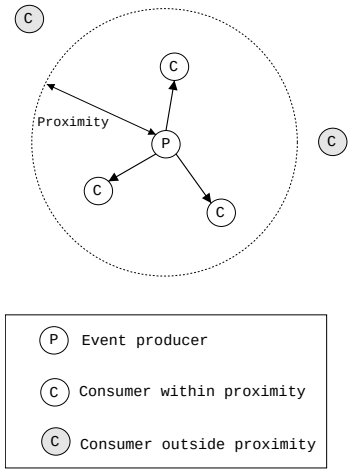

Fig. 4.2: Event dissemination bounded by proximity in STEAM

3. **Proximity** filters define the geographical location in which a particular event type is valid and are specified at the *producer*. The notion of proximity is discussed in further detail in the following section.

Proximity groups

The STEAM communications architecture is based on *proximity groups* [KCM+01], a novel group communication service for wireless networks in which membership is based on both functional aspects and geographical location of participants. A proximity may be of any shape, and may be static, in which case it is attached to a fixed point, or it may be mobile. An example of event dissemination within a circular proximity is illustrated in Figure 4.2.

Proximities are specified by producers for specific event types, and any interested consumer within the specified proximity of the producer will receive notifications of that event type subject to other filters, and assuming network connectivity. Proximities may be specified independently of the producer's physical transmission range, as the underlying proximity group communication service routes messages using a multi-hop protocol [MC03]. The use of proximities to bound event notifications inherently improves system scalability and reduces the use of constrained resources in unnecessary communications.

Limitations of STEAM

Whilst STEAM fulfills the requirement identified for loosely-coupled communication between geographically distributed, ad hoc, mobile, application components, the system does have its own limitations which must be considered. The use of IP multicast communication by STEAM components means that STEAM can only provide best-effort delivery semantics, and thus cannot guarantee that a subscriber will receive a specific event notification , nor that an announcement will be received by nearby devices [Mei03]. Furthermore, multicast flooding of the underlying network by STEAM application components poses a potential scalability issue in resource-constrained mobile, ad hoc networks, although this is mitigated through the incorporation of a routing protocol based on proximity groups to control multicast flooding, as well as a gossip-based optimisation technique [Mei03]. The collocation of STEAM with application components on each mobile device places a potential storage and processing burden on these devices (quantified in section 5.5.1), although in the context of modern mobile storage, the storage burden is not significant[2].

4.2.2 Fulfillment of Requirement 1

The use of the STEAM event service to provide communication between components within the sentient object model addresses the requirement identified for anonymous generative communication. In addition to the advantage offered by the asynchronous style of communication in reducing expensive polling behaviour in resource-poor mobile environments, the event service provides for both extensibility and scalability in the model. Extensibility is aided by the anonymity of communication, permitting new application components to be incorporated without any need for the application to know the identity of all interacting components beforehand. The lack of any centralised components in the event service, in addition to the distributed filtering capabilities, mitigate the disadvantages of multicast flooding and provide for scalability in the model. Specifically, producer-side subject and proximity filters are matched irrespective of the number of consumers in the system, whilst matching of consumer-side content filters is dependent solely on the number of local subscribers [MC03]. The ability

[2]By way of example, the popular iPAQ mobile computer has at least 128 MB of RAM

to filter event notifications based on geographical proximity further aids scalability through localisation of communication.

4.3 Sensor abstraction

Sensors compose the cornerstone of pervasive computing and imbue applications with the ability to perceive their environment. A sensor is defined by [Gov96] as being a device that responds to a physical stimulus, such as thermal energy, electromagnetic energy, acoustic energy, pressure, magnetism, or motion by producing a signal, which is usually electrical. Such hardware transducers are not the only source of data available to a context-aware application, which may also make use of software components that provide information derived from the software environment, such as CPU load, appointments in an electronic calendar, or the estimated remaining lifetime of a battery. Our discussion on *sensors* encompasses both hardware transducers and software sensors.

Traditionally, most developers have taken an application-specific approach to integrating sensors into applications to make them context-aware, e.g., [WHFG92, WSA+95, Rho97, CDM+00]. Following this approach, developers are forced to use low-level, device-dependent protocols in order to interact with the sensor hardware, usually resulting in monolithic systems that do not promote re-use. This approach has the following commonly accepted disadvantages identified in [DSA99]:

1. The task of building a context-aware application is extremely complex, requiring the developer to deal with a range of individual devices and protocols.

2. There is no separation of concerns between application semantics and the acquisition of low-level sensor data.

Furthermore, many applications rely on synchronous polling of sensors to retrieve up-to-date sensor data. Such an approach is inherently non-scalable and not suited to pervasive computing environments where dynamic interactions between very large numbers of devices are envisioned. In addition, this type of interaction requires the application to be aware

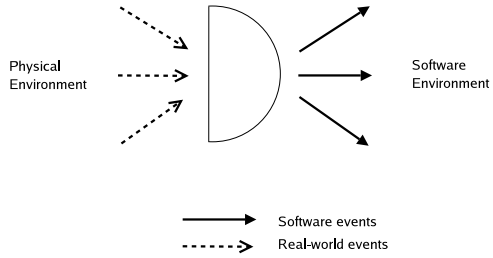

Fig. 4.3: A sensor component

of individual sensor characteristics, such as identity and frequency, in order to implement appropriate polling behaviour. Asynchronous notification of changes in state by sensors provides a more scalable approach to communicating with sensors.

Due to the disadvantages associated with integrating sensors in a tightly-coupled, application-specific manner, we advocate a software abstraction for sensor devices that serves to conceal low-level issues from developers of context-aware applications. We believe such an abstraction offers a crucial separation between the acquisition of sensor data and its use by applications, and provides a uniform interface to interact with sensor data, potentially across multiple applications. A sensor is a software component which encapsulates a hardware sensor device or software sensor, and asynchronously notifies changes in state via typed events. In our model, we overload the term *sensor*, to define it as

> *an entity that* ***produces*** *software events in reaction to a stimulus detected by some real-world hardware device or software component*

A sensor component is illustrated in Figure 4.3, and shows the role of a sensor as an interface between the environment and the software system. It is important to note that the *physical* environment in this respect may include infrastructural or system parameters, as measured by software sensors.

Real-world events

A real-world event may be defined as a change in the state of the real-world, i.e., the physical environment, which is detected by, and causes a change in the state of, a transducer. This definition is adequate for hardware transducers, but does not encompass so-called *software sensors*. We thus extend this definition to include changes in the state of a system as detected by a software component. An example of this might be the detection of user level privileges, current system load, or the latency of a network interface, by operating system software.

Since most transducers contain embedded software to digitize analog input anyway, creating software events, the distinction between real-world and software events is conceptual, but important. As such, a real-world event is an event occurring in the environment external to the application.

Sensor processing

A sensor processes events received from the real world prior to production of software events. The minimal amount of processing required is conversion from the device-specific output of hardware or software components to the software event interface defined by the sensor, that is, the set of event types that the sensor may produce.

Any amount of additional processing may be performed in the sensor to transform the input, for example, to alter the frequency of readings or convert them from a numerical representation to a higher-level symbolic representation. In particular, the sensor may filter incoming real-world events. Filtering incoming data at the sensor reduces unnecessary communication between sensors and other components, and thus contributes to increased scalability. The filtering function of a sensor is especially pertinent in resource-constrained environments, where the reduction in event notifications achieved through filtering provides an associated reduction in the use of the wireless communication interface, and consequent energy savings.

Software events

Software events in the sentient object model are defined as event notifications that conform to the appropriate structure as defined by the event service exploited for communication. A sensor produces software events according to a defined interface.

4.3.1 Fulfillment of Requirement 2

The second requirement we identified in chapter 2 as being essential in the provision of a programming model for the development of context-aware applications is addressed by the sensor abstraction of the sentient object model. The sensor abstraction enables raw sensor data to be represented in an alternative (higher-level, symbolic) format prior to its use by an application, as well as filtering the data, reducing the amount of communication in the network, where bandwidth is limited., and communication expensive. Pre-supposing an existing library of sensor components, application developers are no longer required to write low-level code to interact with sensors. Although the initial development of the sensor components will require the development of such code, the programming tool described in chapter 5 provides a systematic approach to their development, and the resulting components are eminently re-usable. Developers require only a specification of the event types produced by each sensor in order to incorporate them into new applications.

Abstraction of low-level sensors is not unique to the sentient object model, and the approach has been adopted in other approaches to developing context-aware systems, most notably the Context Toolkit [DSA01], where *context widgets* provide a useful abstraction.

4.4 Actuator abstraction

Whilst sensors allow applications to perceive their surrounding environment, actuators provide applications with the means to act upon their environment. The traditional definition of an actuator is the inverse of that of a sensor, i.e., it is a device that responds to an (electrical) signal to produce a mechanical action such as motion, acoustic energy, pressure or thermal energy.

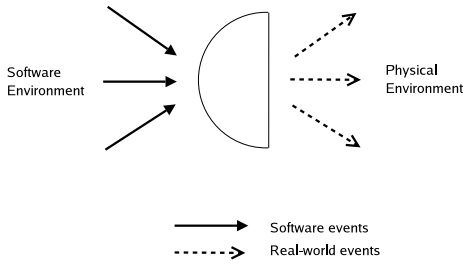

Fig. 4.4: An actuator component

In our model we once again extend the traditional definition of an actuator to encompass a component that effects a change in software, as well as the physical environment. An actuator is a software component which encapsulates a hardware actuator device or software actuator. This leads to the definition of the term *actuator* in our model as

> *an entity that* ***consumes*** *software events, and reacts by attempting to change the state of the environment in some way via some hardware device or software component*

A large proportion of the existing research into context-aware computing has focused almost exclusively on actuation in software and has not dealt with hardware devices. This stems mainly from the historical focus of context-aware computing research as a user-interface technology, using information captured from the environment to dynamically adapt the interface presented to the user. We argue that truly pervasive computing implies more autonomous operation and thus consider actuation as not simply implying the alteration of a user interface.

Software events

An actuator consumes notifications of one or more specific event type, and may perform content filtering on incoming event notifications. The software events consumed by an actuator originate exclusively from sentient objects (see section 4.5), as the result of the evaluation of inference rules.

Actuator processing

Similarly to sensors, the minimal processing required of an actuator is the translation between incoming software events and a device or component-specific protocol. Once again, any amount of additional processing may be performed by the actuator, for example, buffering of input events before production of an output event. Actuators typically perform a translation from symbolic values to numerical, device-specific protocols. Actuators may also filter incoming event notifications based on a range of functional and non-functional attributes. For example, the event service employed in the sentient object model, allows actuators to filter communication based on the geographical proximity of application components.

Real-world events

An actuator produces real-world events in reaction to consumed software events and consequently changes the state of the environment in some way. This may include changing the state of the physical environment via hardware transducers, as well as changing the state of the software environment via software components.

4.4.1 Fulfillment of Requirement 6

The actuator abstraction of the sentient object model addresses the requirement identified for supporting interaction with actuators. This requirement was identified as being important in a programming model for context-aware applications, in order to abstract away from low-level, device-specific protocols. Similarly to sensor components, it is expected that a large collection of actuator components will emerge, eliminating the existing requirement for developers to write low-level, application-specific code to interact with devices. Developers will simply require a specification of the types of event(s) consumed by an actuator in order to control the corresponding device via the production of such events.

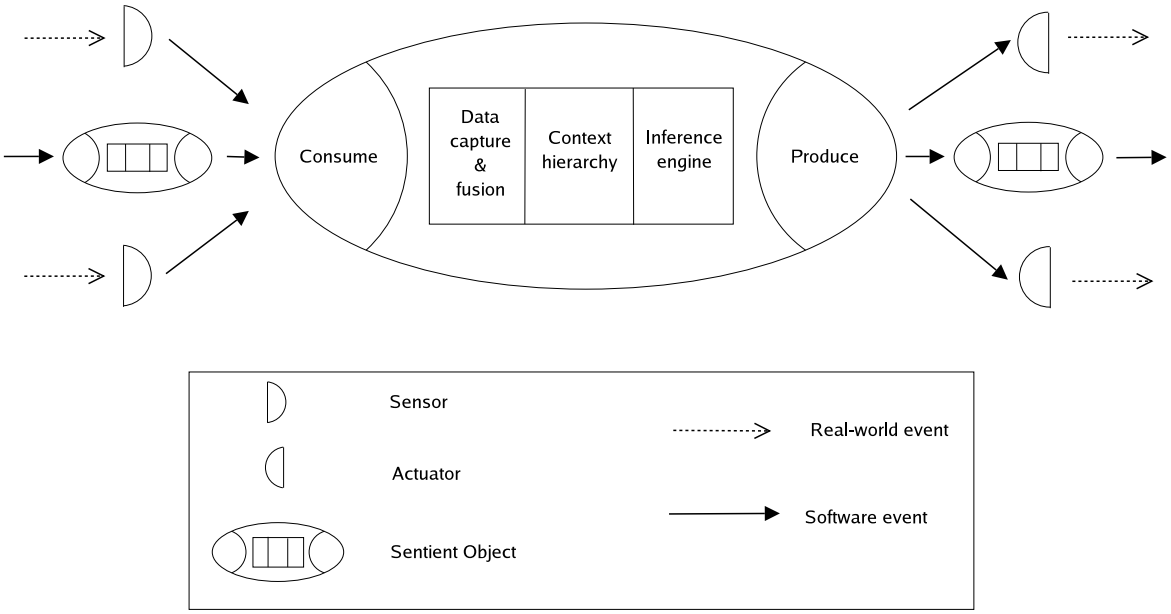

Fig. 4.5: The sentient object model

4.5 Sentient objects

A sentient object is the core component of the sentient object model and is an encapsulated entity, with its interfaces being sensors and actuators. Following our definition of sensor and actuator, we broadly define a *sentient object* as

> *an entity that both consumes and produces software events, and lies in the control path between at least one sensor and one actuator*

We constrain our definition to include the condition that an object must lie in the control path between a sensor and actuator to avoid an overly generic definition that could include other existing software components.

Sentient objects form the major component of a context-aware application, and contain logic that they use to control actuators in a context-aware manner, based on information gleaned from sensors. Internally, a sentient object is composed of three major components, as illustrated in Figure 4.5. The nature of each of these components is discussed in further

detail below.

4.5.1 Data capture and fusion

This component is responsible for the consumption of events produced by both sensors and other sentient objects, and also performs multi-modal sensor fusion to manage the uncertainty of input data and derive higher-level context data from multi-modal data sources. A key consideration within this component is that no distinction is made between event notifications originating from sensor components, and those originating from other sentient objects, with both simply being treated as input data to the sentient object.

Although this component is treated as logically separate from the context hierarchy within a sentient object, in practice the component is closely linked to the context hierarchy, with the active context within the hierarchy influencing the capture and fusion of input data.

Conjunctive content filtering is applied within this component to limit delivery of event notifications to the object, whilst a probabilistic sensor fusion scheme is employed based on Bayesian networks, which provides a powerful mechanism for measuring the effectiveness of derivations of context from noisy sensor data.

Event filtering

A major function of this component of the sentient object is to *filter* incoming event notifications from sensors and other sentient objects, according to the set of active event filters, which in turn depends on the active *context* of the object (see section 4.5.2). The data capture and fusion component defines a set of conjunctive content filters that contain expressions that are matched against the parameter values of event notifications, and notifications are only delivered to the object if the filter matches. The filters are defined for individual event types, when a subscription is made, and may be specified in terms of equality, magnitude, and range operators [Mei03]. The distribution of content filters amongst individual sentient objects, and furthermore, to distinct contexts within each object, significantly aids in the scalability of the model as there are no centralised filtering components, with matching being distributed amongst consumers in the system. The number of filters active and in need of evaluation, on

any particular object is dependent solely on the number of active subscriptions, which in the sentient object model is further constrained by the active context of the sentient object.

The event filtering function of the data capture and fusion component addresses the challenges arising from the generation of large numbers of event notifications by the components of a pervasive computing system, and allows a sentient object to deal only with input data of interest to it at a particular point in time.

Sensor fusion

A major requirement of a programming model for the development of context-aware applications, as identified in section 2.3.8, is support for multi-modal sensor fusion as an approach to managing the uncertainty of readings from a multitude of different sensors, as well as deriving higher level context from lower-level fragments of sensor data. There are a number of possible approaches to the problem of fusing sensor data that have been considered in the sentient object model, and each of these is discussed below.

Multi-variate Gaussian modelling Gaussian modelling may be used to model the probability distribution (and thus sensor noise) of a set of sensor readings according to the following formula

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{4.1}$$

where μ is the mean of the sensor readings, and σ is the variance. A set of such Gaussians may then be used to classify future sensor readings by determining which of the Gaussian functions provides the highest reading when supplied with the senor data.

Multivariate Gaussian modelling is an extension to Gaussian modelling and may be used to model readings from multiple sensors according to the following formula

$$p(\bar{x}) = \frac{1}{\sqrt{(2\pi)^n |C|}} e^{-\frac{1}{2}(x-\mu)^T C^{-1}(x-\mu)} \tag{4.2}$$

where n is the number of sensors, C is the covariance matrix.

Multivariate Gaussians may be characterised for a set of sensors by obtaining the averages and covariances from sample data. These Gaussians are then used classify incoming sensor readings, mitigating uncertainties within the data.

Bayesian networks High-level context information within a sentient object may be modelled by a set of context fragments, derived from inputs from sensors and other sentient objects and represented by events $X_1, X_2, ..., X_n$. The probability of X_1 is defined as the relative frequency with which X_1 occurs in a sequence of n identical experiments.

$$P(X_1) = \frac{n_{X_1}}{n} \tag{4.3}$$

Joint probabilities describe the probability of two (or more) events occurring. If the events are *independent* of each other, the joint probability is given as

$$P(X_1, X_2) = P(X_1)P(X_2) \tag{4.4}$$

Two events X_1 and X_2 are *independent* iff $P(X_1|X_2) = P(X_1)$ or $P(X_1, X_2) = P(X_1|X_2)P(X_2) = P(X_1)P(X_2)$. Complete independence is a very strong and seldom met requirement.

In most cases events are *dependent* on each other. The probability of X_2 *given* X_1 is defined as the *conditional probability* of X_2, given X_1, or $P(X_2|X_1)$. The joint probability is then given as

$$P(X_1, X_2) = P(X_1)P(X_2|X_1) \tag{4.5}$$

This may be rewritten as a general equation for the conditional probability of two events as follows

$$P(X_2|X_1) = \frac{P(X_2, X_1)}{P(X_1)} = \frac{P(X_2)P(X_1|X_2)}{P(X_1)} \tag{4.6}$$

This equation is known as *Bayes' Rule* and in the case of multiple events, this rule is generalized to give the *chain rule*

$$P(X_1, X_2, X_3, ..., X_n) = P(X_1)P(X_2|X_1)P(X_3|X_2, X_1)...P(X_n|X_{n-1}, X_{n-2}, ..., X_1) \tag{4.7}$$

Which may be abbreviated to

$$P(X_1, ..., X_n) = \prod_{i=1}^{n} P(X_i|X_{i-1}, X_{i-2}, X_{i-3}, ..., X_1) \tag{4.8}$$

The chain rule assumes all variables are dependent on each other and is very computationally intensive to compute. In a system with many random variables derived from a range of sensor inputs, the complete specification of a probability distribution requires a very large amount of numbers. For n random binary variables, the complete distribution is specified by $2^n - 1$ joint probabilities. This leads to exponential growth in the model size with associated growth in storage and inference requirements. Bayesian networks allow us to overcome the problem of exponential size by exploiting conditional independence.

Taking 4.7, and the fact that domain knowledge usually allows one to define a subset $pa(X_i) \subseteq \{X_{i-1}, ..., X_1\}$ such that given $pa(X_i), X_i$ is independent of all variables in $\{X_{i-1}, ..., X_1\} \setminus pa(X_i)$ i.e

$$P(X_i|X_{i-1}, ..., X_1) = P(X_i|pa(X_i)) \tag{4.9}$$

Then

$$P(X_1, ..., X_n) = \prod_{i=1}^{n} P(X_i|pa(X_i)) \tag{4.10}$$

So, the joint probability $P(X_1, X_2, ..., X_n)$ can be implicitly represented as the conditional probabilities $P(X_i|pa(X_i))$ for $i = 1, ..., n$.

A *Bayesian network* may be described as the representation of a joint probability distribution defined on a finite set of discrete random variables, as a directed, acyclical graph (DAG). The nodes in a Bayesian network represent propositional variables of interest and the (directed) links represent informational or causal dependencies among the variables. Dependencies between nodes in the network are quantified through the specification of conditional probabilities for each node given its parents, and the network supports the computation of probabilities of any subset of nodes, given evidence about any other subset [PR03]. Formally, a Bayesian network is defined by [ZR97] as a triplet (N, E, P), where N is a set of nodes, E is a set of edges where $E \subseteq N \times N$, and P is a set of probabilities. Each node is labelled by a value v_i, and each variable takes a particular value from a discrete domain and is assigned a vector of probabilities, $P(v_i)$. Each element of $P(v_i)$ represents the belief that v_i will take a particular value. A Bayesian network is a DAG such that a directed edge $e = < s_i, t_i > \in E$

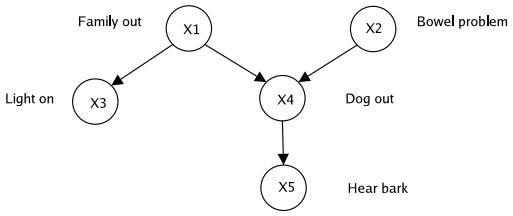

Fig. 4.6: An example Bayesian network

indicates causal influence from source node s_i to target node t_i. For each node t_i, the strength of causal influence from its parent s_i are quantified by a conditional probability distribution $p(t_i|s_i)$, specified in an $m \times n$ matrix, where m is the number of discrete values possible for t_i and n is the number of values for s_i.

A classic example of a Bayesian network given in [Eug91] is illustrated in Figure 4.6, and allows the determination, with bounded probability, of the value of individual parameters, given observation of other parameters. The network describes the causal relationships between whether a family is out ($X1$), whether their dog has a bowel problem ($X2$), whether the light in their home is on ($X3$), whether the dog is outside ($X4$), and whether the dog can be heard to bark ($X5$). The links between the nodes represent dependencies between values, and the network represents that a family may[3] put their dog outside if they go out, or it is experiencing a bowel problem. The dog may bark if it is put outside, and finally, the family may leave the light on if they go out. Through observation of the values of the leaf nodes (is the dog barking, is the light on), probabilities may be calculated for the values of the other nodes in the network (what is the likelihood the family is out).

Bayesian networks operate through propagation of beliefs through the network following the assertion of evidence about the existence of certain entities. Taking the example of the network in Figure 4.6, our belief in whether a family is out of their home is influenced by observing whether the light is on in their home. Similarly, our belief in whether the light is on is not directly influenced by whether we hear the dog barking, so there is no direct link between node $X5$ and node $X3$ of the network.

[3]With a bounded probability

Dempster-Shafer theory Dempster-Shafer theory provides an approach to representing plausabilities and is considered a generalised Bayesian theory (see section 4.5.1) whereby probabilities are assigned to sets of possibilities rather than individual events. Dempster-Shafer differs from Bayesian theory in three major ways: [HM93]:

1. Evidence is represented as a Shafer belief function rather than a probability density function. All possible mutually exclusive context fragments are enumerated in a frame of discernment Θ. For example, a sensor may detect whether a door is open or closed, giving

$$\Theta = \{open, closed\} \tag{4.11}$$

Other, multi-modal, sensors may contribute their own observations, by assigning their beliefs over Θ. As [WSSY02] states, this assignment function is known as the probability mass function of the sensor S_i, denoted by m_i. The probability of a sensor reading, is then indicated by a confidence interval

$$[Belief_i(A), Plausability_i(A)] \tag{4.12}$$

Wu states that the lower bound of this interval is the belief confidence, accounting for all evidence E_k that supports the given proposition [WSSY02]

$$Belief_i(A) = \sum_{E_k \subseteq A} m_i(E_k) \tag{4.13}$$

The upper bound is then the plausibility, accounting for all observations that do not rule out the proposition

$$Plausability_i(A) = 1 - \sum_{E_k \cap A = \phi} m_i(E_k) \tag{4.14}$$

2. Any two belief functions may then be combined into a new belief function using Dempster's rule of combination. Following the explanation provided by Wu, a reading m_i provided by sensor S_i may be combined with a reading m_j, provided by sensor S_j as follows

$$(m_i \oplus m_j)(A) = \frac{\sum_{E_k \cap E_{k'} = A} m_i(E_k) m_j(E_{k'})}{1 - \sum_{E_k \cap E_{k'} = \phi} m_i(E_k) m_j(E_{k'})} \tag{4.15}$$

3. Prior odds are not required for the computation of evidence for a proposition, as Dempster-Shafer theory assumes ignorance in the absence of prior belief.

Dempster-Schafer theory has previously been proposed as a generalizable sensor fusion solution for context-aware computing [Wu03].

Bayesian networks for sensor fusion in sentient objects

Whilst multi-variate Gaussian modelling is useful for fusing the output of multiple, mono-modal sensors, it does not provide an elegant approach to the problem of fusing the output of multiple, multi-modal sensors and managing the uncertainty of such sensors and is not considered further.

Both Dempster-Shafer theory and Bayesian networks provide an approach to fusing the output of multi-modal sensors, and dealing with the uncertainties of sensor data. As [HM93] note, Bayesian networks provide a clear and well understood method for incorporating how the likelihood of a possibility is conditioned on another event, whilst conditioning mechanisms in Dempster-Shafer are less clear. Since the determination of the context of a sentient object depends on the determination of the likelihood of particular state based on multiple uncertain sensor inputs, the application of Bayesian networks afford the preferable choice within the sentient object model. Furthermore, within the sentient object model conditioning is easy to extract through probabilistic representation of sensor data, and prior odds are available. Demspter-Shafer theory is more applicable to situations where uncertainty cannot be assigned probabilistically. Following this reasoning Bayesian networks have been selected as the mechanism to provide multi-modal sensor fusion in sentient objects.

We can construct a Bayesian network to fuse multi-modal sensor data, by representing the evidence received from multi-modal sensor readings as leaf nodes, and the higher-level context information we need to derive from the sensor readings, as a root node, in a two-level Bayesian network. The uncertainty of each sensor is represented as a conditional probability table associated with the causal relationship linking the sensor nodes to the node representing the context information. This is illustrated in Figure 4.7 where a set of sensors which produce evidence (E) in the form of multi-modal readings, jointly contribute to the hypothesis (H)

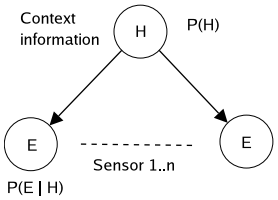

Fig. 4.7: A Bayesian fusion network

that a piece of context information takes on a particular discrete value. The uncertainty of each individual sensor is captured as a conditional probability table in each sensor node, calculated *a-priori*.

Given prior experimental data, it is possible to calculate the probability that we obtain specific evidence E from sensors, given a known hypothesis H, or $P(E|H)$. For example, we may perform an experiment which determines the probability that a range-finding sensor gives a reading indicating an obstacle at 10 cm, when we have placed an obstacle at 10 cm.

What is not straightforward to calculate, and what the sensor fusion component of a sentient object requires, is the probability that a hypothesis H is true, given specific evidence E obtained from sensor readings, or $P(H|E)$. Or following our example, the probability that there *is* an obstacle 10 cm in front of our sensor given that the sensor gives a reading indicating that there is. This is known as *abduction*, or prediction of the antecedent given the consequent.

In order to construct a Bayesian network to fuse fragments of context data into higher level context information in a sentient object, the following steps are followed:

1. Add a root node X_r representing the fused context information, and specify the vector of probabilities P_{v_r} for the possible values of this node.

2. Choose the set of relevant context fragments (provided by sensor events) represented by nodes $X_i, ..., X_n$ that contribute to the fused context information.

3. While there are fragments left:

   (a) Pick a fragment X_i and add a node to the network for it

(b) Set $pa(X_i)$ to some minimal set of nodes already in the net such that the conditional independence property is satisfied $P(X_i|X_{i-1}, ..., X_1) = P(X_i|pa(X_i))$. Following the definition of a fusion network as being a two-level Bayesian network, in practice, the network is constructed such that $pa(X_i) = X_r$ for all i where $1 \leq i \leq n$

(c) Define the conditional probability table $p(X_i|X_r)$

At runtime, the nodes representing fragments of context information are updated with evidence provided by the delivery of input events, and the root node representing the fused context information may be queried, providing the current probability that it has a particular value.

Example

By way of example, consider an application based on a sentient object which determines the identity of the user of a sentient couch. The identity of the user may be considered high-level context information which may be used by a sentient object to control application behaviour in a context-aware manner, for example, by proactively setting user preferences on an audio player, or by routing telephone calls to the telephone extension nearest the couch.

Within this scenario, there are a number of sensors that may be used to determine identity with different levels of certainty, and by fusing the outputs of these sensors, the application is able to determine the identity of an individual with a bounded probability. The sources of data available to the system consist of (1) an iButton[4] chip carried by all people, with an associated reader at the entrance to the room containing the couch; (2) an RFID chip embedded in each individual user's coffee mug, with an associated reader near the couch; and (3) a sentient object which reports the identity of a user based on the weight detected on the couch. There are a number of potential sources of uncertainty inherent in each of these sensors, precluding reliance on any single sensor. Users are not required to press their iButton prior to entering the room; not all users will carry their coffee mugs at all times; a user's weight may fluctuate between uses of the couch. Each sensor reports the identity of

[4]http://www.ibutton.com

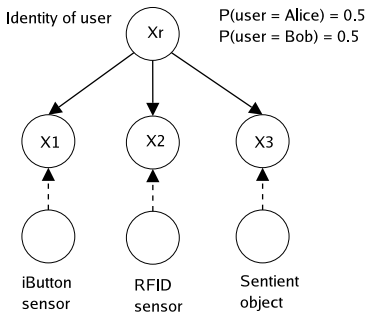

Fig. 4.8: An example fusion network fusing the output of three sources of identity data

the user it detects, and the fusion network measures the probabilities that a specific user is on the couch.

Constructing the fusion network consists of the following process, resulting in the network illustrated in Figure 4.8.

1. The root node, X_r, of the network is added, with a vector of probabilities P_{v_r} representing the belief that a particular user is on the couch. The vector has n entries, where n is the number of users of the couch.

2. The relevant context fragments are the outputs of each of the three sensors, each of which reports a user identifier, or null if no user is identified by the sensor.

3. Three nodes are added to the network, representing the output of each of the three sensors, $X_1 ... X_3$. For each of the nodes:

   (a) A directed link is added from X_r to X_i, so that $pa(X_i) = X_r$

   (b) The conditional probability table $p(X_i|X_r)$ is specified. The conditional probabilities are derived from experimentation, giving the probability that each individual sensor gives the correct reading, for known input.

The conditional probability table for node $X1$ is illustrated in Table 4.1 and captures the probability that the identity reported by the iButton sensor is correct. Possible sources of uncertainty arise from the fact that users may not press their iButton chips to the reader,

$X1$	Xr	
	Alice	Bob
Alice	0.63	0.44
Bob	0.01	0.02
Unidentified	0.36	0.54

Table 4.1: Conditional probability table for Node $X1$ in Figure 4.8

the reader may not register the chips, or a person may use a chip which is not registered in their name.

At runtime, nodes X_1, X_2, and X_3 are periodically instantiated with evidence received from the respective sensors they represent. Each time evidence is received from a sensor, in the form of an event notification, the set of probabilities that X_r takes on a certain value are recalculated using Bayesian formulae, and the hypothesis with the highest probability is asserted as a fact with the relevant probability that it is true.

Fulfillment of Requirement 3

The data capture and sensor fusion component of a sentient object, as implemented based on Bayesian networks, fulfills the third requirement identified of a programming model for context-aware applications. The use of a probabilistic sensor fusion scheme provides a powerful and efficient approach to fusing multi-modal sensor readings providing fragments of context data, allowing the determination of higher-level context. The incorporation of context-specific, content-based event filters in the component effectively reduces the overall number of event notifications delivered to the object, contributing to the efficiency of the fusion process.

A further, qualitative, advantage offered by the data capture and fusion component, is in the provision of a systematic approach to limiting input and fusing data within a sentient object. The provision of a standard approach to multi-modal sensor fusion which may be applied to a wide range of application scenarios, eases the development process. By limiting the types of input that need to be dealt with at any point in time through the abstraction of an active context, the specification of sensor fusion networks is further eased (see section 4.5.2 for discussion of the relation between sensor fusion and the context hierarchy). Furthermore, by limiting Bayesian fusion networks to two-levels, a common approach to specifying fusion

services across a range of applications is provided.

A potential drawback inherent in the use of Bayesian networks for the fusion of sensor data, is the requirement to gather extensive a-priori probability data to construct the network.

The development of the data capture and fusion component is further simplified through the notion of a context as defined in the next section. By decomposing the function of the data capture and fusion component into a set of distinct contexts, the development of complex sentient objects is broken down into a set of more manageable steps.

4.5.2 Context hierarchy

The overall context of a sentient object is represented as a set of discrete logical facts, following a logic-based approach to context representation (see section 4.5.3). Following this approach, fragments of context data derived from input events are stored as *facts* within working memory of a rule-based inference engine, as described in section 4.5.3.

Multi-modal context fragments are fused by the sensor capture component to determine higher level context information (also stored as facts), which is subsequently used to determine the overall *context* of the object. In terms of the context hierarchy, a context is defined as a distinct state in which the object may exist at a point in time, determined based on inputs from sensors and other sentient objects, and defining a set of appropriate behaviours, or actions to be carried out. This definition mirrors that provided in section 2.2.3, in that a context is defined based on environmental inputs, and in turn, defines appropriate actions. The set of contexts in which a sentient object may exist is represented as a hierarchy, inspired by the Context-Based Reasoning (CxBR) paradigm [GA99], which aims to provide conciseness and simplicity of representation and consequent efficiency of computation. This paradigm derives its name from the hypothesis that the actions taken by an entity are highly dependent on the entity's current context, i.e., a recognized situation defines the set of appropriate actions to be taken, and the identification of future situations is simplified if all possible actions are limited by the current situation itself.

The hierarchical representation of a set of contexts within a sentient object is based on the observations that (1) only a subset of all available inputs are typically necessary to recognise

and treat the key features of a situation; (2) in any given situation, there are a small number of well-defined actions that take place; and (3) the presence of a new situation will typically require alteration of the present course of action. Representation of the states in which an object may exist as a set of distinct, hierarchically linked contexts allows the object to treat only the relevant subset of all available inputs, whilst links between contexts define transitions between states.

The context hierarchy component thus encapsulates knowledge about actions to be taken and possible future situations into a set of distinct contexts. Each context within the hierarchy is a specific situation in which the sentient object may exist during its lifetime and is defined by the following:

1. The set of **input event types** that are of interest and are consumed within the context. This is typically a subset of all the events consumed by the object. In this way a context reduces the amount of input an object needs to deal with at any point in time.

2. A set of **filters** limiting delivery of event notifications to the set of input events defined for the context.

3. A **fusion network** to fuse input events and derive high-level context information.

4. The set of **production rules** relevant in that context. Rules control both the behaviour of the object within the context, as well as the transition between contexts. Rules may also be applied to fuse fragments of context data. There are thus three distinct types of rules in a context:

   (a) Behavioural rules encode knowledge as to how to treat the situation and control behaviour whilst the context is active.

   (b) Transition rules monitor whether the context is still active according to input events, and control the de-activation of the context and transition to other contexts.

   (c) Fusion rules are optional within a context and control the fusion of fragments of context data derived from input events.

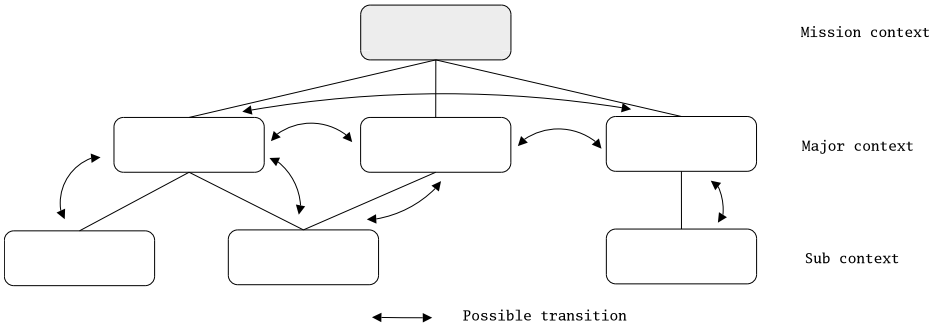

Fig. 4.9: The context hierarchy

5. The set of **transition contexts** to which a transition may be made from the context.

6. The set of **output events** produced in the context.

In addition, only a subset of all *facts* representing the context of a sentient object are relevant in each context. This is implicit, since the rules within a context will only depend on a subset of facts.

The structure of the context hierarchy is illustrated in Figure 4.9, illustrating the three levels of the hierarchy, and allowable transitions between contexts. There are three different types of context, defined as follows,

1. A **mission** context is always active during the sentient object lifecycle and defines inputs that are of interest to the object throughout its lifecycle, and rules that always valid. There is only one mission context per object.

2. A **major** context defines distinct *strategic* objectives of the sentient object, that assist in fulfillment of the goals of the object. Although multiple major contexts may be defined within the context hierarchy of a single object, only one is active at any point in time. Each major context is linked to one or more child sub contexts. A major context is active for as long as input events indicate it is a valid context. During this time, a major context will typically activate sub contexts to carry out specific actions.

3. A **sub** context defines *operational* actions carried out in fulfillment of a major context,

of which it is a child. Up to one sub context may be active at any point in time, and sub contexts are typically only active for a short period of time. A sub context is generally activated, causing a set of rules to fire, and then deactivated straight away.

The hierarchical representation of the set of potential contexts in which a sentient object may exist is motivated by a number of distinct advantages arising from such a representation. The primary advantage gained is a reduction in complexity of sentient object development, through provision of a systematic approach to dealing with context. The context hierarchy links the other logically separate internal components of a sentient object, since both event filters and rules governing object behaviour are specified at the level of a context. The task of developing a sentient object is ameliorated to the development of a set of contexts and the relationships between them, with input event filters, a fusion network, behavioural and transition rules, and output events making up a context. Specifically, the use of a three-level hierarchy of contexts was determined by the fact that any intelligent behaviour consists of a set of strategic objectives which evolve over time, with each strategic objective being made up of a set of actions necessary to achieve that objective. Each atomic action is related to one or more of the strategic objectives, has clearly defined start and end points, and takes place over a relatively short period of time. The strategic objectives are always relative to an overall mission, giving rise to the three-level hierarchy. Whilst future development might obtain value in extending the hierarchy to further levels, or indeed reducing the number of levels, the current decomposition of a mission being composed of a set of strategic objectives which in turn are composed of a set of atomic actions proves the most useful.

A further advantage, and that which motivated the original hierarchical approach to context representation, is a runtime advantage arising from the partitioning of the rule base according to the active context. Since each context defines the set of rules relevant in that context, only a subset of the overall rule-base is active at any point in time, according to the currently active context. By reducing the number of rules that need to be evaluated, the performance of the inference engine may be increased. Associated with the partitioning of the rule-base is the reduced likelihood of conflict between rules in a potentially large rule-base.

The context hierarchy also contributes to the proactive behaviour of sentient objects. The

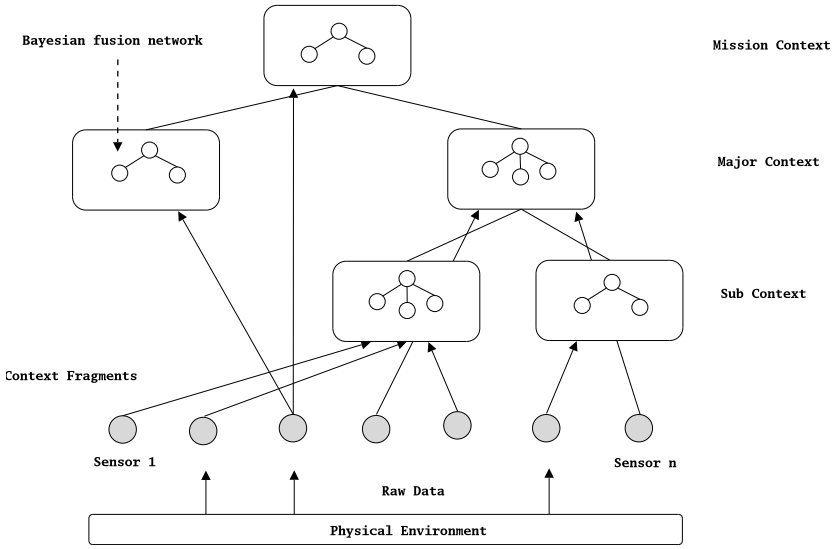

Fig. 4.10: Sensor fusion in the context hierarchy

hierarchy captures the possible transitions between individual contexts, giving each context an expectation of future states of the system and allowing it to act in relation to these future states.

Sensor fusion and the context hierarchy

The fact that only a small portion of sensory input is relevant at any point in time according to the event filters of the active context, is used to enhance the effectiveness of the probabilistic sensor fusion scheme described in section 4.5.1, by limiting the number of nodes necessary in a sensor fusion network in each context.

Sensors in the sentient object model are highly distributed components, with dynamically changing configurations due to the mobility of sentient objects. In addition, the set of input events relevant at a particular point in time is highly dependent on the active context at that time. The data capture and fusion component takes advantage of this fact to perform fusion at the level of a context within the context hierarchy. A context defines which input events are relevant to that context as well as those which must be monitored in order to detect when transition to another context is indicated, and only this set of events is eligible for fusion. A

Bayesian network is specified within each context in order to fuse the fragments of context information obtained from input events. Rather than one monolithic fusion network per object, smaller networks may be constructed in each context, based only on the set of input events which are relevant within the context. The integration of Bayesian fusion networks into the context hierarchy is illustrated in Figure 4.10. This figure shows how each context in the hierarchy is only interested in a subset of the sensor input, and Bayesian network fragments within each context act to fuse the context fragments obtained from the relevant input events.

Fulfillment of Requirement 4

The context hierarchy component fulfills the fourth requirement identified of a programming model supporting the development of mobile, context-aware applications, that is support for context representation, at two levels. At the lowest level, context information is represented within a sentient object using a set of facts stored in the working memory of a rules-based inference engine. Storage of context information as facts provides a simple, flexible, and extensible approach to representing data derived from multi-modal sensors and other sentient objects.

At a higher level, the context hierarchy defines the set of situations in which an object may exist as distinct contexts, encapsulating appropriate fusion mechanisms, behaviours and expectation of future states. The context hierarchy plays a vital role in easing the development of sentient objects and the applications based on them, by providing a manageable approach to defining context-aware behaviour and linking the data capture and fusion and inference engine components, exposing a structured approach to the development of both.

4.5.3 Inference engine

The third and final internal component of a sentient object provides a mechanism for reasoning intelligently about context information represented by the context hierarchy. Intelligent reasoning capabilities are added to a sentient object by way of the inference engine component which encapsulates a knowledge base in the form of a set of production rules. The

knowledge base specifies a set of pre-defined actions which are triggered when the object is in a particular context, in other words sentient objects follow an *Event-Condition-Action* execution model [dInK01]. The motivation behind using a production rule-based inference engine within a sentient object is due to the following considerations

- Production rules provide a natural approach and uniform structure to representing knowledge within a system, enabling sentient behaviour. Appropriate knowledge engineering techniques exist to derive appropriate rules in application domains [Lio90].
- Context information derived from input events is stored as a set of logical facts. These facts may be used to update the working memory of a production rule-based inference engine, providing a close coupling between context representation and aiding in the efficiency of inference.

The sentient object model exploits the built-in pattern-matching capabilities of CLIPS[5] (C Language Integrated Production System), a production rule-based system created by NASA in the mid-1980s and now in popular use in both industry and academia. CLIPS employs *forward chaining* rule activity whereby known facts cause rules to fire, which in turn causes the assertion of further facts, and causes further rules to fire. CLIPS has a number of distinct advantages [dIn02] that make it suitable for inference in sentient objects. Specifically, CLIPS is based on the Rete [C.L82] algorithm, containing an efficient mechanism to solve the many-to-many pattern matching problem, complex sets of relations may be expressed in patterns, and the language is integrated with a number of popular programming languages, including C++ and Java.

Context representation

Context information is represented in the sentient object through the assertion of logical facts within working memory of the inference engine. There are three types of facts together representing the overall context of a sentient object.

[5] http://www.ghg.net/clips/CLIPS.html

1. Atomic facts represent individual event notifications produced by sensors or other sentient objects. For example, an atomic fact may be asserted to represent the reading of an ultrasonic distance sensor mounted on a vehicle.

2. Fused facts represent higher-level context information derived from atomic facts by a fusion network, and each fact has a probability associated with it that it is correct. For example, a fused fact may be asserted to represent that there is a 70% chance that there is an obstacle located in front of a vehicle, given a set of atomic facts.

3. Custom facts represent fragments of context information which are unique to a particular application, and may make use of custom functionality implemented for the application. For example, a fact may be asserted that represents the bearing between two co-ordinates, as calculated by a set of navigational formulae implemented outside working memory.

Atomic and fused facts are updated through delivery of event notifications to the sentient object, whilst custom facts are updated by event delivery, or via functionality implemented in code outside working memory. The representation of context data as facts within working memory of the inference engine eases the development of rules which reason about context, as fragments of context may easily be matched within production rules.

Inference and the context hierarchy

The rules within the knowledge base of the sentient object are modularised according to the context (in the context hierarchy) in which they are relevant. The main advantage gained by assigning rules to individual contexts is in the provision of a systematic approach to the development of the knowledge base of a sentient object. Rather than having to specify a single, monolithic knowledge base, a sentient object's knowledge base is developed incrementally, as a set of contexts, each of which only deals with a subset of all available input. In this way, development of the knowledge base is vastly reduced in complexity, as is the probability that unexpected behaviour arises due to conflicting rules.

A further advantage gained from the context hierarchy at runtime is in efficiency of inference. The performance of rule-based systems can suffer as the number and complexity of rules in the knowledge base increases. The context hierarchy within the sentient object mitigates this performance decrease by making only a subset of the knowledge base active at any time. This follows from the hypothesis that there are only a limited number of actions that can realistically take place in any situation, and limits the number of rules active and in need of evaluation at any one point in time.

Event production

The *actions* taken by a sentient object as the result of the evaluation of context information by the inference engine consist of the production of events for consumption by actuators and other sentient objects. Sentient objects thus act as event producers, and it is through event production that the sentient object is able to interact with the environment. Both functional and non-functional event filters may be specified by the sentient object on event notifications that it produces.

Fulfillment of Requirement 5

The inference engine component of the sentient object, based on and leveraging the inference capabilities of a production rule-based system, fulfills the fifth requirement identified of a programming model for context-aware applications. The rules-based approach to intelligent inference was selected for sentient objects due to a number of considerations. Most importantly, the use of rules provides a natural way to capture and represent knowledge, that is used to drive the proactive behaviour of an object, increasing the accessibility of the programming model to developers. Although closely linked, the actual knowledge base is clearly separated from the code controlling the capture, and fusion of context data, allowing easy changes and enhancements to be made to the rules without affecting the sentient object code. Rules provide a uniform and modular representation of knowledge, that can be used to encode formal rules and heuristics with ease. Finally, the rule-based inference engine still performs when faced with incomplete, uncertain, and fuzzy inputs.

4.5.4 Developer support

The final requirement we identified as critical to the development of a programming model for mobile, context-aware applications, is the provision of an accessible, and usable development environment which exposes the support offered by the previous six requirements to the application developer. A unified approach is required which exposes full lifecycle support for design, implementation, testing, and maintenance of sentient objects, to the developer in an intuitive manner. Specifically, our aim is to reduce the low-level syntax required to develop mobile, context-aware applications based on the sentient object model. We propose a domain specific language for the development of mobile, context-aware applications, in the form of a graphical programming tool with associated code generator.

Fulfillment of Requirement 7

The next chapter describes the implementation of a graphical programming tool providing a set of domain specific abstractions that may be applied by developers to compose mobile, context-aware applications. This tool fulfills the requirement identified in Chapter 2 for the provision of developer support.

4.6 Summary

This chapter described the sentient object model, a programming model for the development of context aware applications in mobile ad hoc environments. The sentient object model fulfills the first six requirements of such a programming model, as identified in section 2.3.8. The next chapter describes a graphical programming tool based on the sentient object model and fulfilling the seventh and final requirement of the programming model that is designed to significantly ease application development, making it available to a wider audience.

Chapter 5

A Programming Tool for Mobile, Context-Aware Applications

This chapter describes a graphical programming tool for mobile, context-aware applications that was developed based on the sentient object model. The major goal of the tool is to provide support to the application developer in using the sentient object model for the design and development of mobile, context-aware applications. The tool enables rapid prototyping of applications and generates low-level procedural code, obviating the need for the developer to write it.

The current version of the tool generates Java code and uses an implementation of the STEAM event service, but also creates independent descriptions of components that may be used as templates to generate implementations of sentient objects in different languages and using different APIs.

5.1 Implementation considerations

The programming tool itself is implemented using the Java 2 SDK, version 1.4, this choice being motivated by a number of advantages of the Java language. The most compelling motivation is that pure Java is completely platform independent and portable across multiple systems both at the source and binary level. Since the goal of the programming tool is to

make the development of mobile, context-aware applications accessible to as wide an audience as possible, its implementation in a platform-independent language greatly facilitates this. The tool will run unchanged on any operating system platform with Java support, including Solaris, Linux, and Microsoft Windows. In addition to Java's platform independence, the language comes with a rich class library offering good support for collections, XML manipulation, and more, whilst excellent Java APIs exist for working with Bayesian networks, rule-based inference engines, and graphical interfaces. In particular, the Swing toolkit offers a rich set of lightweight components that can be used to rapidly create effective user interfaces.

The programming tool produces as output, automatically generated code implementing a sentient object specification defined by the application developer, as well as language-independent descriptors for sensors, objects, and actuators. The object specification, although developed using a Java-based interface, is essentially independent of any individual programming language. The programming tool is thus potentially able to produce code in any language in which the object could be implemented, including C++ and Java. The first version of the programming tool generates code for sentient objects in Java, with the following motivation:

1. Java provides a truly ubiquitous platform on which to deploy pervasive applications.

2. Comprehensive Java library support exists to support the components of sentient objects.

3. The performance of modern Java virtual machines is increasingly comparable to C / C++ code [JN04].

5.1.1 Event middleware

As discussed in section 4.2.1, the sentient object model uses a distributed event service for wireless ad hoc networks, incorporating a proximity-based programming model. The first version of the programming tool is designed to use a Java implementation of the STEAM event service, namely jSTEAM. This version of STEAM was ported from a C++ implementation to Java by members of the Distributed Systems Group.

5.2 Sensor development

The first task in context-aware application development is the specification of sources of context data, known as *sensors* in the sentient object model. In the model, sensors are simply defined as producers of software events, and consist of software components that perform a transformation between device specific protocols and a common event notification dialogue. In addition, each sensor has an associated *descriptor* that describes its event interface, that is the number and type of events and their parameters, that the sensor produces.

Since sensor development is highly dependent on individual hardware transducer devices and specific protocols, it is not practical to support visual development of these components, rather supposing their pre-existence. In order to ease development of sensor components, the programming tool offers a `Sensor` base class which may be extended by sensor components. This class, illustrated in Listing 5.1, provides a set of methods to specify the proximity in which sensor event notifications are valid, to publish event notifications, and to instantiate the event service middleware with location data provided by either simulated GPS co-ordinates, or the output of a real GPS receiver.

```
public class Sensor{
  protected S_Steam steam;
  protected SP_SteamProducerEntity steamProducer;
  protected SP_Shape proximity;

  public Sensor(SP_Shape sensorProximity){}
  public void announceEventType(SP_dsEventType eType){}
  public void publishEvent(SP_dsEvent eventInstance){
  public void startSensor(int range, //fixed location
                int period, double latitude, double longitude){}
  public void startSensor(int range, //use a simulated location array
                int period, double[] latitude, double[] longitude){}
  public void startSensor(int range, int period){} //use real GPS data
}
```

Listing 5.1: `Sensor` base class

5.2.1 Sensor descriptor

Each sensor has an associated descriptor, which is an XML document describing the set of events produced by a sensor, and the type and name of each parameter of each event

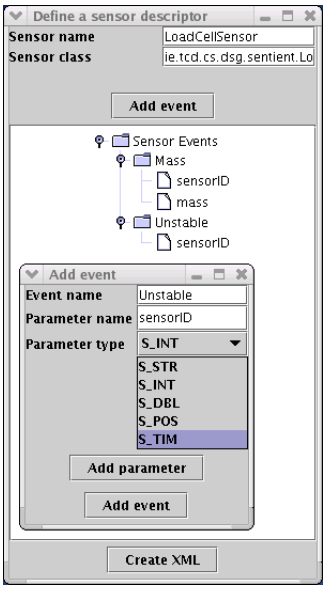

Fig. 5.1: Support for sensor descriptor definition

produced. The DTD[1] describing the form of a sensor descriptor is illustrated in Listing 5.2.

```
<?xml version='1.0' encoding='us-ascii'?>
<!-- DTD for a sensor definition file"-->
<!ELEMENT sensor (sensor-name, sensor-class, event+)>
<!ELEMENT event (event-type, parameter+)>
<!ELEMENT parameter (param-name, param-type)>
<!ELEMENT sensor-name (#PCDATA)>
<!ELEMENT sensor-class (#PCDATA)>
<!ELEMENT event-type (#PCDATA)>
<!ELEMENT param-name (#PCDATA)>
<!ELEMENT param-type (#PCDATA)>
```

Listing 5.2: DTD for an XML sensor descriptor

The important elements of the sensor descriptor file are:

1. A unique name for the sensor.

2. The fully qualified class name of the sensor component.

[1]Document Type Definition

3. A list of events produced by the sensor. Each event contains a list of parameters of the event, specifying

   (a) The name of the parameter.

   (b) The STEAM *type* of the parameter.

Visual support is offered by the programming tool for the development of the XML sensor descriptors, and is illustrated in Figure 5.1. The tool permits the developer to specify the name and classname of a sensor, and to add event definitions to the descriptor before generating the XML descriptor file.

5.3 Actuator development

The development of elements which act upon the environment, known as *actuators* in the sentient object model is similar to sensor development, albeit consisting of a transformation in the opposite direction - from software events to low-level, device-specific protocols. Visual tool support is again not offered for the development of actuator components, giving the developer the necessary flexibility to work with low-level protocols. Once again, the development of sentient objects themselves supposes that a library of actuator components exists for a set of hardware and software devices, and application developers will not be required to develop these components from scratch. The programming tool provides an `Actuator` base class which is extended when developing specific actuator components. This class contains an overloaded method which starts the event service using different location parameters, as well as a method to subscribe to a specific event type. In addition, an abstract class `ActuatorDeliveryCallback` is provided, defining a single method which is implemented to handle event delivery. It is within this method that the transformation from a software event to a real-world event (e.g., a serial RS-232 port event, or a software command) is made. The class and interface are illustrated in Listing 5.3.

```
import ie.tcd.cs.dsg.jsteam.*;

public abstract class ActuatorDeliveryCallback extends SC_CallbackDelivery{
  public abstract void deliver(SC_dsEvent e); //handle an event notification
}

public class Actuator{
  protected S_Steam steam;
  protected SC_ConjunctiveContentFilter contentFilter;
  protected ActuatorDeliveryCallback deliveryCallback;
  protected ActuatorAnnouncementCallback announcementCallback;
  protected SC_SteamConsumerEntity steamConsumer;

  public void subscribe(String eventType){}//subscribe to an event type
  public void startActuator(int range, int period, double latitude,
              double longitude, ActuatorDeliveryCallback dCallback){}
  public void startActuator(int range, int period, double[] latitude,
              double[] longitude, ActuatorDeliveryCallback dCallback){}
  public void startActuator(int range, int period,
              ActuatorDeliveryCallback dCallback){}
}
```

Listing 5.3: `Actuator` base class and `ActuatorDeliveryCallback` base class

5.3.1 Actuator descriptor

In the same way that each sensor has an associated descriptor file that describes the events and their associated parameters produced by the sensor, each actuator has an associated descriptor file which describes the type of events consumed by the actuator. Similar support is offered in the programming tool for the visual development of actuator descriptors, to that of sensor descriptors, allowing the specification of the name and type of events consumed by the actuator. The actuator descriptor DTD is illustrated in Listing 5.4.

```
<?xml version='1.0' encoding='us-ascii' ?>
<!-- DTD for an actuator definition file"-->
<!ELEMENT actuator (actuator-name,actuator-class,event+)>
<!ELEMENT event (event-type,parameter+)>
<!ELEMENT parameter (param-name,param-type)>
<!ELEMENT actuator-name (#PCDATA)>
<!ELEMENT actuator-class (#PCDATA)>
<!ELEMENT event-type (#PCDATA)>
<!ELEMENT param-name (#PCDATA)>
<!ELEMENT param-type (#PCDATA)>
```

Listing 5.4: DTD for an XML actuator descriptor

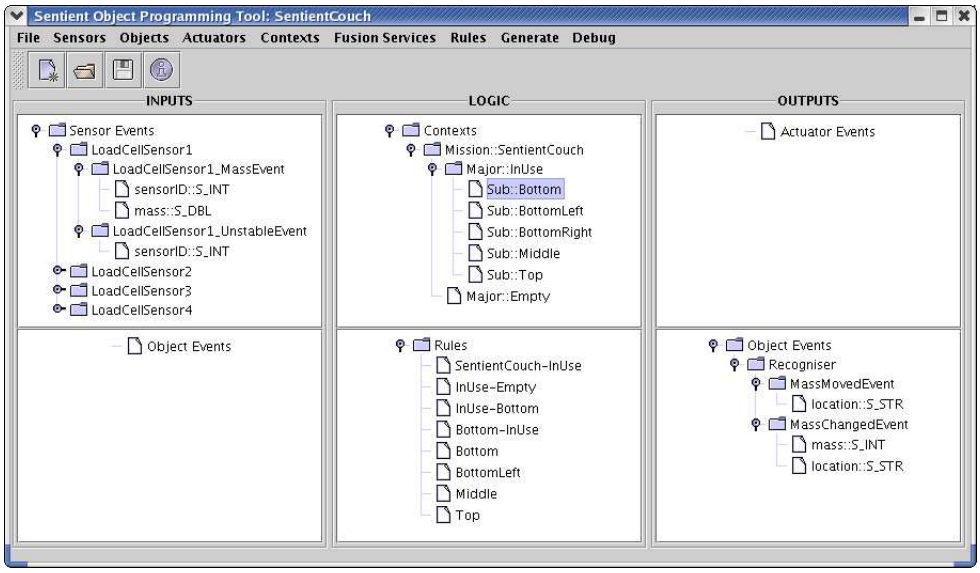

Fig. 5.2: The main interface to the programming tool

5.4 Sentient object definition

The main functionality offered by the programming tool is support for the development of sentient objects. Following the definition of sentient objects as being both consumers and producers of software events, and lying in the control path between sensors, actuators and other sentient objects, the tool provides support to define input events, output events, and internal sentient object logic.

The main interface to the programming tool is illustrated in Figure 5.2, showing the three main aspects to object specification, moving from left to right in the figure.

5.4.1 Input events

The definition of input events is the first step in the development of a sentient object. Sentient objects consume events produced both by sensors and by other sentient objects, but all event notifications are in the common event notification dialogue of STEAM. A sentient object does not therefore distinguish between sensor events and object events, although this distinction is made in the input specification of the programming tool, to allow the developer to use both

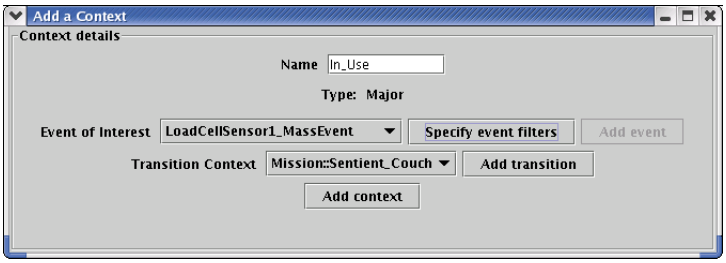

Fig. 5.3: Context definition screen

sensor descriptors and object descriptors to specify inputs.

Input events are defined by adding sensor and/or object descriptors within the tool, providing the object with event type information which is used to filter and consume event notifications. Content filters are then specified on input events at the level of a context in the context hierarchy, as outlined in the next section.

5.4.2 Context hierarchy

The next step in programming a sentient object is the specification of the hierarchy of contexts in which the object may exist over the course of its lifetime. The main aspects of a context to be defined, are (1) the set of events that are of interest within the context; (2) a set of filters defined on the events of interest; (3) the set of contexts to which the context may transition; (4) a fusion network to fuse multi-modal input events; and (5) the set of rules valid within the context. The set of events in which a context is interested, and which will be delivered whilst the context is active, and the set of transition contexts, are defined by the developer using the screen illustrated in Figure 5.3. The main component within this screen is a drop down box listing all input event types available to the object (as specified by descriptors added to the objects inputs), with no distinction made between those published by sensors, and those published by other sentient objects.

Individual event types of interest may be selected from the drop down box, and a filter defined over the parameters of each event type using a screen as illustrated in Figure 5.4. This screen presents the developer with a list of the parameters of the selected event type,

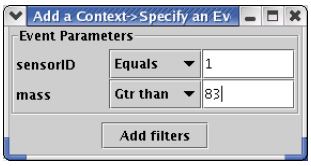

Fig. 5.4: Event filter definition screen

with a drop down box containing the type-specific filter operators for each parameter.

Finally, the set of allowable transition contexts may be specified by selecting the contexts from a drop down box containing all contexts within the hierarchy. To allow specification of transitions to contexts which have not yet been defined (since major contexts are defined before their constituent sub-contexts), an edit facility is provided so transitions to newly defined contexts may be added.

5.4.3 Sensor fusion network

Following definition of the context hierarchy, the next major step in development of a sentient object, is the definition of sensor fusion services on a *per context* basis. The rationale behind specifying sensor fusion services per context is provided in section 4.5.2, and is based on limiting the size of a Bayesian fusion network, as well as decomposing the complex task of specifying fusion services into more manageable steps.

The current version of the programming tool provides support for the definition of sensor fusion services based on Bayesian networks. A network may be specified for each context, and may fuse any event data consumed within the context. Graphical support for specification of fusion networks is offered, greatly easing the task of the developer, and obviating the need to write low-level syntax to describe the network.

The fusion network specification screen is illustrated in Figure 5.5. The first step in defining the network is selection of the context with which the network is associated, from a drop down box containing a list of all contexts defined for the object. Once a context is selected, another drop down box is populated with all input event types available in the context. The developer is then able to select a specific event type, populating a drop down box

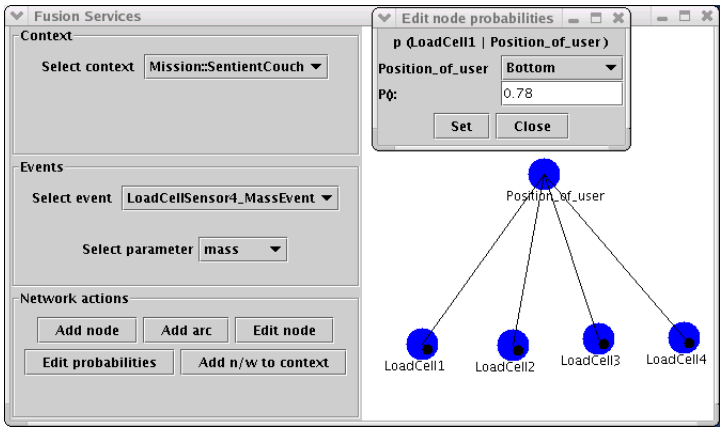

Fig. 5.5: Fusion network specification

containing the parameters of the event type. A node representing a parameter of a specific event type may then be added to the network through a simple drag and drop maneuver. Alternatively, a node representing a fused value may also be added to the network.

Once all nodes have been added to the network, a conditional probability table must be specified for each node in the network, using the probability editor screen, also illustrated in Figure 5.5.

5.4.4 Output events

The specification of the types of events the sentient object can produce may be carried out at any point in the object development, before definition of the inference rules. Output events, however, must be defined before inference rules since the rules may cause event production and thus the type of events that may be produced by the object must be defined.

The process of specifying event types that may be produced by the object is similar to the specification of input events discussed in section 5.4.1. A sentient object may produce software events which are consumed either by actuators, or by other sentient objects and although there is no differentiation between the production of actuator events and object events, a distinction between the two is exposed in the programming tool. This distinction is not strictly necessary, but allows the developer to see at a glance where the event will be

consumed.

Event types which may be produced by the sentient object are thus specified by adding actuator and/or object descriptors that provide information about the type of events to which the target actuator or object subscribes.

5.4.5 Inference rules

The final step in the development of a sentient object is the definition of the *knowledge base* of the object, in the form of a set of production rules. During object execution, this set of rules reasons about the object's working memory, composed of a set of *facts* continuously updated by the delivery of event notifications.

The programming tool supports the development of two major types of rules, although other ad hoc types of rules may be added by the programmer.

Transition rules

Each individual context has a set of transition rules, which govern the transition from the context to other contexts according to the structure of the context hierarchy. The number of transition rules depends both on the type of context, and its position in the hierarchy. Each sub-context has x associated transition rules, where x is the number of major contexts which have a link to the sub-context. A sub-context transition rule governs transition to the relevant parent major context, on completion of the appropriate actions associated with the sub context.

Each major context has $x+y$ transition rules, where x is the number of child sub-contexts which the major context has links to, and y is the number of other major contexts in the hierarchy to which the major context has a link. The single mission context in the context hierarchy has x transition rules where x is the number of major contexts in the hierarchy.

The transition rule specification screen is illustrated in Figure 5.6. A rule is developed by firstly selecting the context *from* which, and the context *to* which the transition is made, from drop down boxes. Finally, the conditions under which the transition occurs are defined, by specifying conditions on facts in the working memory. These facts represent context data,

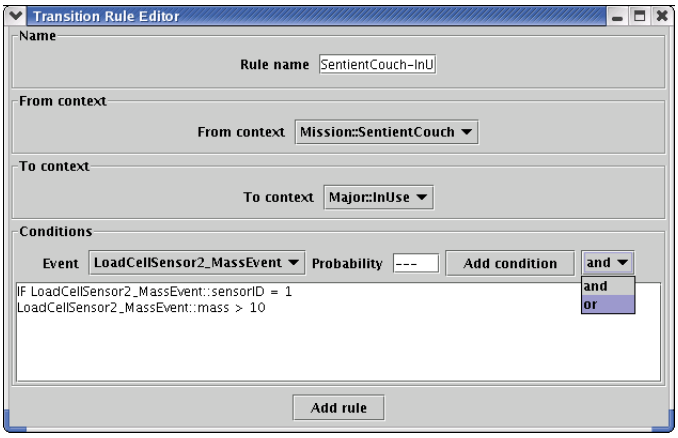

Fig. 5.6: Transition rule definition screen

either as atomic or fused events. If the selected fact is fused context data, a probability value may be specified. This refers to the certainty that the fused fact has the specified value, and the rule will only evaluate to true if the probability associated with the fact is equal to, or greater than, this value.

Behavioural rules

Behavioural rules are so-called because they define the behaviour of a sentient object, with *behaviour* defined as the production of software events for consumption by actuators or other sentient objects. The screen provided by the programming tool for the definition of behavioural rules is illustrated in Figure 5.7.

This screen allows the specification of a behavioural rule as consisting of an *action* to be taken when a defined *condition* is met. Conditions are specified as current context values, that is the value of context fragments obtained from sensor events, or higher level context information fused from such fragments. After providing a unique name for the rule and selecting what context the rule is valid in, a drop down box provides a list of all context fragments and fused context information available within that context. Individual fragments may be selected, and combined with the `and` or `or` operators. Furthermore, it is possible to specify the probability of fused context fragments.

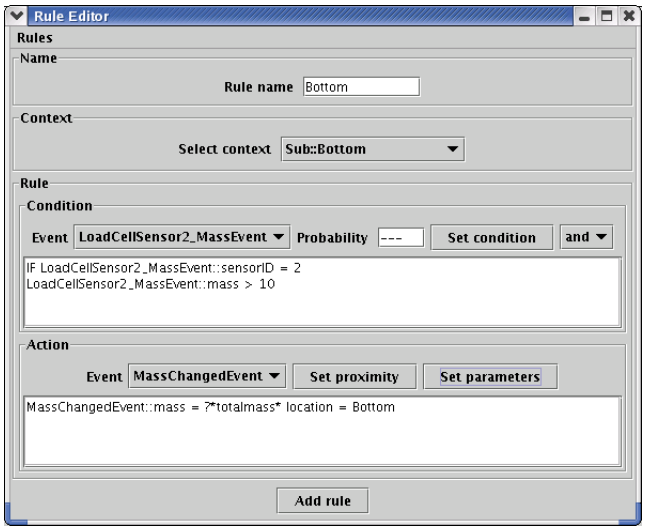

Fig. 5.7: Behavioural rule definition screen

Actions are specified as the production of an event(s) when the condition(s) is satisfied. A drop down box provides a list of all possible event types which may be produced by the object, allowing a specific event type to be selected. The proximity in which the event will be produced, as well as event-type specific parameters may also be set.

5.4.6 Object descriptor

Each sentient object developed using the programming tool is described by an XML descriptor file. This descriptor fully describes the sentient object, including inputs, contexts, fusion networks, rules, and outputs, and enables load/save capability for editing object specifications within the tool. As a generic description of a sentient object, it also acts as a template for code generation, and enables portability between code generators implementing the object specification in different implementation languages. The DTD for a sentient object descriptor is listed in Appendix A.

5.5 Code generation

Once the specification of a sentient object has been completed using the visual components of the programming tool, low-level code is generated which implements the object specification. Whilst it is possible to generate the object code in potentially any language, the first version of the programming tool generates Java code, as motivated in section 5.1.

The majority of the generated Java code extends from a set of abstract classes providing base functionality, or implements defined interfaces. The major components of the generated code are illustrated in Figure 5.8 and are discussed in the following sections.

5.5.1 Sentient object

A sentient object is generated that extends from the `SentientObject` class illustrated in Figure 5.8. Following the sentient object model, the main attributes of a sentient object are a collection of contexts in which the object may exist, an inference engine, and an instance of the STEAM event service to provide for event-based communication.

As discussed in section 4, sentient objects employ an inference mechanism based on production rules, and specifically the production rule system CLIPS [NAS99]. Since the original CLIPS implementation is in C, there were number of alternatives available to incorporate CLIPS rules and inference capabilities into generated sentient objects. One alternative was to use Java Native Interface (JNI) code from within the sentient object, to access the CLIPS library. Following this approach, the JNI-CLIPS bridge could be generated from scratch, or the capabilities of an existing library could be reused. JClips [2], developed by Maarten Menken at Vrije University in the Netherlands provides an excellent example of a reusable JNI interface to the native CLIPS libraries, that was a candidate for inclusion in generated sentient objects.

The other alternative, is to use a pure Java implementation of a production rule-based system based on CLIPS. Examples of such systems include OPSJ [3], developed by Production Systems Technologies (PST), a company founded by Dr. Charles Forgy, the inventor of the

[2] http://www.cs.vu.nl/ mrmenken/jclips/
[3] http://www.pst.com/opsj.htm

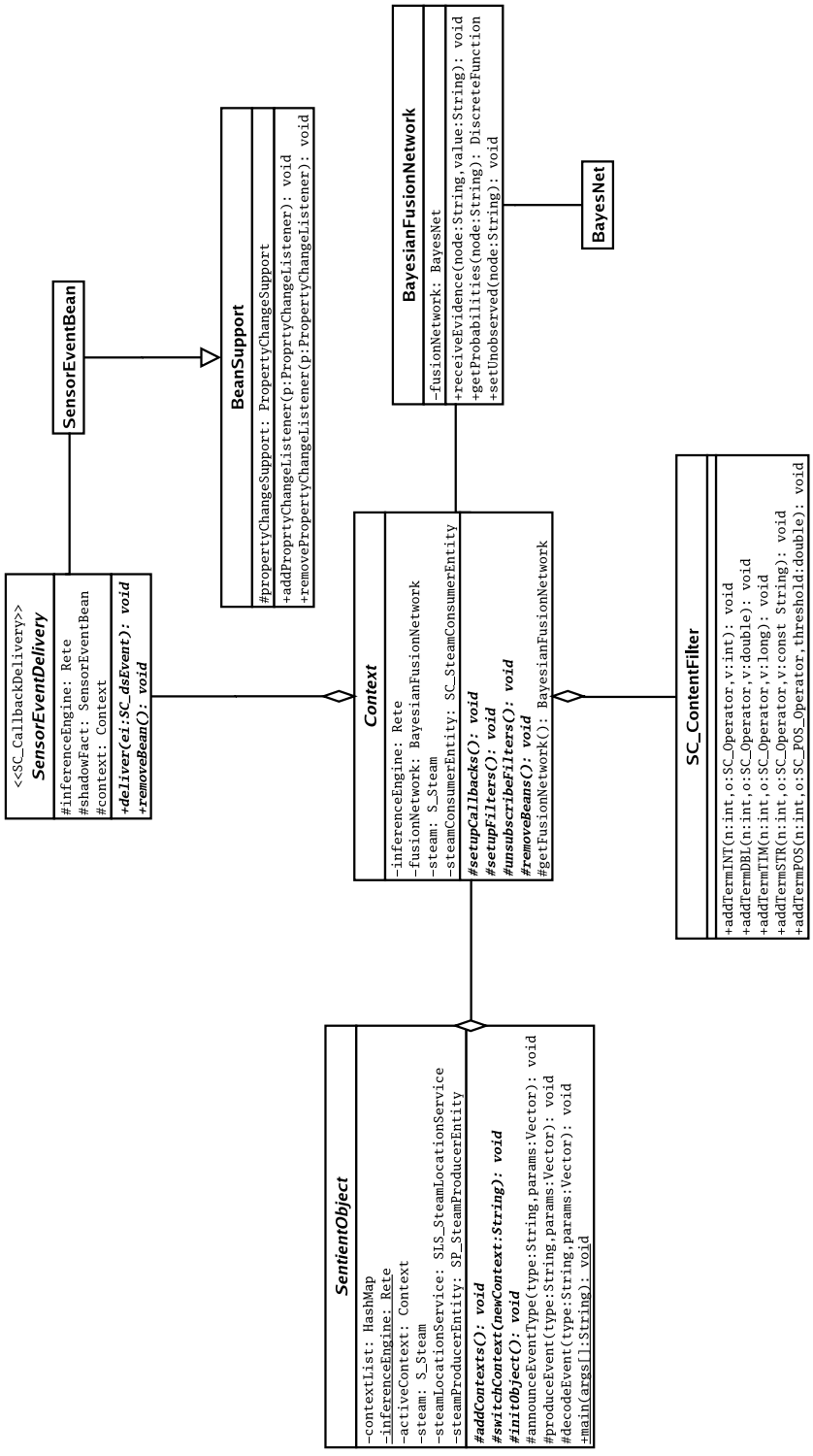

Fig. 5.8: Major components of a generated sentient object

Rete algorithm. A further alternative was provided by the Java Expert System Shell (JESS) [FH03], which is a pure Java implementation of CLIPS, tightly integrated with the Java programming language.

The choice was made to use JESS to provide the inference capabilities in generated sentient objects. This choice was motivated primarily by the fact that it is a pure Java implementation and provides useful integration with the language not achieved through simply providing a JNI interface to native libraries. In addition, Jess is freely available for academic use, whilst OPSJ is not publicly available, and the licensing costs not clear. Furthermore the Jess libraries provide a compact footprint (471 KB) suited to resource poor devices.

The inference engine component of the object is an instance of `jess.Rete` which is essentially an instance of Jess. This class maintains a working memory of facts and a set of rules which act upon facts to derive inferences. One aspect of the tight integration of Jess with Java is the use of JavaBeans[4] to provide a connection between working memory and the Java application, in a mechanism known as *shadow facts* [FH03]. The generated code makes use of shadow facts to enable event notifications to dynamically update working memory, as explained in section 5.5.2.

The generated code uses the STEAM event-based middleware to facilitate communication between application components, and more specifically a Java implementation of STEAM. As discussed in section 4.2.1, STEAM was selected to provide event based communication between distributed, mobile ad hoc application components due to its inherently distributed architecture, as well as the ability to geographically scope event dissemination amongst location-aware application components. The Java version of STEAM used in the generated sentient objects is suitably compact, requiring libraries of 407 KB in size.

Each generated sentient object has an instance of STEAM, as well as an instance of the STEAM location service providing location-awareness to application components. Furthermore, each sentient object has a reference to the *active* context, defined as the context that is active at a particular point in time. The active context controls the delivery of event notifications according to a set of filters, fusion of context data, and inference of appropriate behav-

[4] http://java.sun.com/products/javabeans/

iour. When a context-switch is indicated by a transition rule, the active context is switched by the `switchContext()` method. Since a sentient object is able to produce events, each object also has methods to announce event types, as well as an instance of `SP_SteamProducerEntity` to produce event notifications.

5.5.2 Contexts

Each sentient object has a set of contexts in which it may exist, with each individual context controlling the delivery of event notifications to the object according to a set of filters, as well as providing for the fusion of fragments of context data, and the inference of behaviour according to active rules. Each context is generated as a subclass of the `Context` class illustrated in Figure 5.8.

Event consumption

Event notifications are consumed and delivered to the sentient object according to the set of event content filters specified during the context definition. Each context maintains a set of content filters as specialisations of the type `SC_ContentFilter` and a set of delivery callbacks, one per type of event consumed by the object. The delivery callback for each event type has a reference to a Java bean representing that event type, and providing a link to the working memory of the inference engine.

Each context has methods to set up event filters and delivery callbacks upon activation, as well as methods to unsubscribe from event notifications and remove filters on context de-activation.

Context representation

Sentient objects employ a combination of the *logic-based* and *object-oriented* approach to context representation, meaning that fragments of context information are stored as *facts* represented by objects, within the working memory of the object's inference engine. The generated object code specifically uses the Jess mechanism of dynamic shadow facts to enable working memory to be dynamically updated by event delivery. A Java bean is generated

which represents each type of event that may be consumed by the object. Each bean has a property (a private variable, with associated accessor and mutator) representing each of the parameters of the event type. An example of a generated Java bean is illustrated in Listing 5.5, for an event of type **Mass**, with two parameters, *SensorID: int* and *mass: double*. It can be seen that the bean inherits from the `BeanSupport` class and each time a bean property changes, an event is fired, notifying the inference engine of the new parameter value, and causing the fact representing this event to be dynamically updated in working memory.

```
public class LoadCellSensor1_MassEvent_Bean extends BeanSupport {
  private int sensorID;
  private double mass;

  public void setSensorID(int newSensorID) {
    int old_sensorID = sensorID;
    sensorID = newSensorID;
    propertyChangeSupport.firePropertyChange("sensorID",
      new Integer(old_sensorID),
      new Integer(sensorID));
  }

  public void setMass(double newMass) {
    double old_mass = mass;
    mass = newMass;
    propertyChangeSupport.firePropertyChange("mass",
      new Double(old_mass),
      new Double(mass));
  }

  public int getsensorID() {return sensorID;}
  public double getmass() {return mass;}
}
```

Listing 5.5: A generated Java bean representing an event of type **Mass**

The delivery callback class for each event type maintains a reference to the appropriate Java bean for that event type, and updates the bean properties upon event delivery. A generated delivery callback class for the event type represented by the Java bean in Listing 5.5 is illustrated in Listing B.2.

From this listing it can be seen that the constructor method of the callback instantiates a new bean which it then registers with the inference engine object, through calls to `defclass()` and `definstance()`. The subsequent delivery of an event instance is handled by the `deliver()`

method that extracts the values of the event parameters and updates the relevant bean properties, in turn updating facts within working memory. In this way, the delivery of event notifications dynamically updates context fragments represented as rules in the inference engine.

```
import ie.tcd.cs.dsg.jsteam.*; import jess.*;

public class LoadCellSensor1_MassEvent_DeliveryCallback
 extends SensorEventDelivery {
  private LoadCellSensor1_MassEvent_Bean loadCellSensor1_MassEvent_Bean;
  private Context context; //the active Context
  private Rete inferenceEngine; //the inference engine component
  private BayesianFusionNetwork fusionNetwork; //the fusion network

  public LoadCellSensor1_MassEvent_DeliveryCallback(Rete ie, Context ctxt) {
    super();
    context = ctxt;
    fusionNetwork = context.getFusionNetwork();
    inferenceEngine = ie;
    loadCellSensor1_MassEvent_Bean = new LoadCellSensor1_MassEvent_Bean();
    try { //register the bean with JESS
      inferenceEngine.defclass("LoadCellSensor1_MassEvent_Bean",
        "loadCellSensor1_MassEvent_Bean", null);
      inferenceEngine.definstance("LoadCellSensor1_MassEvent_Bean",
        loadCellSensor1_MassEvent_Bean, true);
    }
    catch (Exception e) {e.printStackTrace();}
  }

  public void deliver(SC_dsEvent ei) { //handle delivery of an event instance
    int sensorID_intVal = ei.parValINT(0); //extract the first event parameter
    //update the shadow fact and fusion network
    loadCellSensor1_MassEvent_Bean.setsensorID(sensorID_intVal);
    fusionNetwork.receiveEvidence("sensorID_intVal",""+sensorID_intVal);
    double mass_dblVal = ei.parValDBL(1); //extract the second event parameter
    //update the shadow fact and fusion network
    loadCellSensor4_MassEvent_Bean.setmass(mass_dblVal);
    fusionNetwork.receiveEvidence("mass_dblVal",""+mass_dblVal);
  }
}
```

Listing 5.6: A generated delivery callback for event of type **Mass**

5.5.3 Sensor fusion

Associated with each context is a Bayesian sensor fusion network which serves to fuse fragments of context data received from sensors into higher level information, as well as manage

uncertainty inherent in the context fragments. Each context object has an attribute of type `BayesianFusionNetwork`, which in turn has an attribute representing the Bayesian network itself, as illustrated in Figure 5.8.

The programming tool makes use of the EBayes engine for embedded Bayesian networks in the sentient objects it generates. The EBayes engine was developed by Fabio Cozman at CMU explicitly for the needs of embedded devices. As a result it provides an extremely compact footprint (26 KB), and has limited memory requirements due to dynamic linking behaviour. EBayes was selected over other Bayesian network implementations for Java due to these low resource requirements. For example, Netica-J [5], a competing Bayesian network offering for Java requires 977 KB of storage for its set of library files. Furthermore, EBayes is distributed under the terms of the Gnu Public License (GPL)[6], in comparison to the significant commercial license costs associated with Netica.

The Bayesian network is represented as a Java class file in the format prescribed by the EBayes engine[7], and reflecting the design made by the developer in the visual network builder. An example network generated in this format and representing the two nodes X_r and X_1 of Figure 4.8 in section 4.5.1, is illustrated in Listing 5.7.

The generated sentient object uses the EBayes engine to calculate the posterior marginal distribution of a particular node in the fusion network. Methods are provided within the generated `BayesianFusionNetwork` class to update the network based on events consumed by the sentient object.

5.5.4 Inference rules

The knowledge base of a sentient object consists of a set of production rules generated based on the specifications made using the visual rule development components. Sentient objects generated by the programming tool leverage the inference engine provided by Jess, and thus rules are generated in Jess syntax, which is itself based on CLIPS syntax.

As discussed in section 5.5.2, context is represented in a sentient object by logical facts

[5] http://www.norsys.com/netica-j.html
[6] http://www.gnu.org/copyleft/gpl.html
[7] http://www-2.cs.cmu.edu/ javabayes/EBayes/index.html/

```
import BayesianNetworks.*;

public class FusionNetwork extends BayesNet {
  public FusionNetwork() {
    DiscreteVariable user_identity =
      new DiscreteVariable("user_identity",
        DiscreteVariable.CHANCE,
        new String[] {"Alice", "Bob"}
      );

    DiscreteVariable iButton_sensor =
      new DiscreteVariable("iButton_sensor",
        DiscreteVariable.CHANCE,
        new String[] {"Alice", "Bob"}
      );

    DiscreteFunction p1 = //define conditional probability table for node
      new DiscreteFunction(
        new DiscreteVariable[] {user_identity},
        new DiscreteVariable[] {},
        new double[] { 0.5, 0.5 }
      );


    DiscreteFunction p2 = //define conditional probability table for node
      new DiscreteFunction(
        new DiscreteVariable[] {iButton_sensor},
        new DiscreteVariable[] {user_identity},
        new double[] {0.63, 0.44, 0.37, 0.56}
      );

    add( new DiscreteVariable[] {user_identity, iButton_sensor});
    add( new DiscreteFunction[] {p1, p2});
  }
}
```

Listing 5.7: A generated fusion network in EBayes format

asserted in the working memory of the inference engine. Specifically, the objects generated by the programming tool make use of the shadow fact mechanism provided by Jess, to link event delivery to dynamic updates of facts in working memory. Each fragment of context information, as well as context information fused by the Bayesian fusion network, is represented by a Java bean object which in turn provides a shadow fact in working memory. Inference rules are thus able to reason about context information dynamically updated in working memory through event delivery.

All rules generated by the programming tool are written to a single text file per sentient

object, named `controller.clp`. The separation of object inference logic into a separate text file, distinct from sentient object class files is based on the following advantages:

1. Rules are not compiled, but loaded at runtime, so may be changed without the need for re-compilation of sentient object code.

2. Sentient object logic is concentrated in a single location and de-coupled from application code, so may be easily modified without affecting other application components.

3. Rules are specified *declaratively*, at a high level of abstraction, and are semantically clear to visual inspection.

An example of a behavioural rule generated by the programming tool is illustrated in Listing 5.8. This example shows how shadow facts representing context fragments, as well as facts representing fused context fragments may be used in the LHS of the rule. The RHS of the rule then asserts further facts, or can cause event production by calling back to the sentient object. The rule in Listing 5.8 checks the readings from four individual sensor against a fused reading to determine whether or not to perform a context transition.

```
(defrule InUse-BottomLeft
?ac <- (active-major-context InUse)
?cu <- (context-updated true)
?aw <- (average-weight (averageweight ?X))
(SmartCouch_LoadCellSensor1_MassEvent_Bean (sensorID 1)(mass ?LoadCellSensor1_MassEvent))
(test (> ?LoadCellSensor1_MassEvent ?X))
(SmartCouch_LoadCellSensor2_MassEvent_Bean (sensorID 2)(mass ?LoadCellSensor2_MassEvent))
(test (< ?LoadCellSensor2_MassEvent ?X))
(SmartCouch_LoadCellSensor3_MassEvent_Bean (sensorID 3)(mass ?LoadCellSensor3_MassEvent))
(test (< ?LoadCellSensor3_MassEvent ?X))
(SmartCouch_LoadCellSensor4_MassEvent_Bean (sensorID 4)(mass ?LoadCellSensor4_MassEvent))
(test (< ?LoadCellSensor4_MassEvent ?X))
=>
(retract ?ac)
(retract ?cu)
(bind ?*weight* ?*totalweight*)
(assert (active-sub-context BottomLeft))
)
```

Listing 5.8: Example of a generated behavioural rule

Temporal validity of context data

As noted by [dIn02], events are discrete temporal occurrences, and the frequency of event sources differs between event sources. Context data derived from sensor events and fusion networks, are represented by shadow facts within working memory, obviating the need for

continuous garbage collection of facts, as described in [dIn02]. A shadow fact is dynamically stored in working memory for each fragment of context data received by the sentient object, whether derived from atomic events, or as a result of fusion. Shadow facts rely on JavaBeans, generated by the programming tool.

The use of shadow facts ensures that a fact representing a context fragment is only asserted in working memory when the context fragment it represents changes. There is therefore no need to garbage collect facts through explicit retraction. A shadow fact is valid for one execution cycle of the inference engine, and then is removed from working memory until the shadow fact is subsequently updated. If a rule matches a fact representing a fragment of context data, the rule will only fire if the fact has been updated since the last time the inference engine executed. This means that inferences are not made on stale context data, and rules will only fire if the facts they match on have been recently updated.

5.5.5 Object descriptor

In addition to generating low-level Java code implementing the sentient object specification made in the programming tool, the tool also automatically generates an XML descriptor of the object, as described in section 5.4.6.

5.5.6 Runtime flow of control in a generated object

The flow of control in a generated sentient object at runtime is illustrated in Figure 5.9, and explained below:

1. Event notifications are consumed according to the set of filters defined by the active context of the sentient object.

2. A delivery callback is made for each type of event notification received. The delivery callback

   (a) Updates the relevant node in the fusion network of the active context with the evidence contained in the event notification.

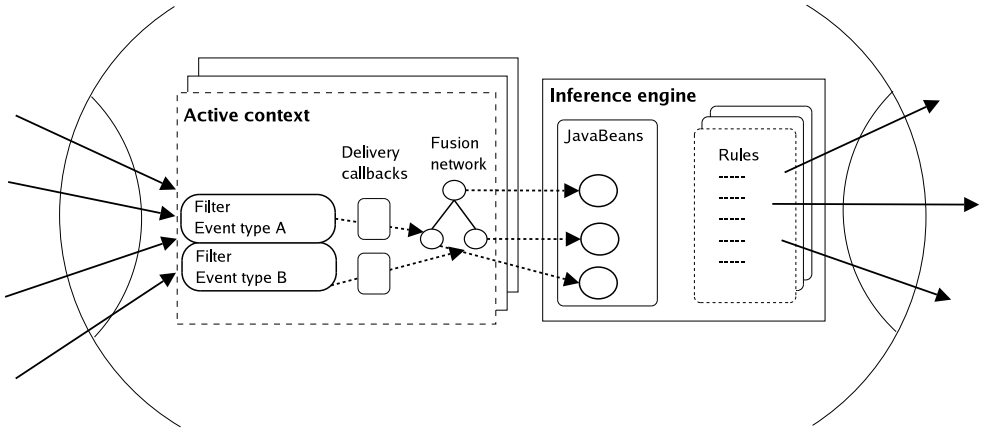

Fig. 5.9: Flow of control at runtime in a generated sentient object

(b) Updates the Java bean representing the event type as a shadow fact in working memory of the inference engine.

3. The inference engine executes the set of rules for the current active context.

4. Any context transition or event notification indicated by the execution of the inference engine is carried out.

Sentient objects are single-threaded, with actions controlled by the currently active context.

5.6 Code generation within the sentient object model

The automated generation of sentient object code by the programming tool offers a number of advantages with regard to the development of mobile, context-aware applications, whilst at the same time suffering from potential drawbacks.

5.6.1 Advantages

The primary advantage of having low-level code generated by the programming tool is that application developers only have to work with high-level, domain-specific abstractions, and do

not need a deep understanding of the implementation language. This leads to a reduction in programming errors, as well as making prototyping easier through a reduction in development time. The code generated by the programming tool offers the following specific advantages over hand-crafted implementations:

Quality

The quality of automatically generated code is directly related to the quality of the generating code. By increasing the quality of the generating code at a single point, the quality of an entire application code-base is increased. Bugs may also be fixed and new features introduced at a single point in the generating code, thereby making generated code more robust, as well as more flexible and agile.

Consistency

Generated sentient object code has consistent variable naming and API design [Her03], in marked contrast to code engineered by hand.

Productivity

Since code generation significantly reduces the time spent on implementation, there is consequently more time available in the development lifecycle for application design. In addition, since the actual process of writing the application code is negligible, development effort may be concentrated on ensuring that the generating code is of sufficient quality.

Abstraction

The application design is captured in an abstract form external to the application code (for example in an XML file). This enables the application to be easily ported to another implementation platform, by way of an alternative code generator, which is significantly more efficient than porting equivalent hand-crafted code. In addition, products supporting the application may be generated from the abstract design, including documentation or test cases.

Customisable

The code generator maintains a *single point of knowledge* [Her03], capturing how an application is implemented. By parameterising the generation process appropriately, applications could be easily customised. The code generated by the programming tool places the entire rule-base in a single file which may be easily edited by hand to further customise the application. Furthermore, the rules are written in a high-level procedural language that is more easily understood.

5.6.2 Drawbacks

Although automated code-generation by the programming tool offers significant advantages in terms of the sentient object model, a number of drawbacks remain, primarily related to a loss of control over the code by developers:

Limited flexibility

Generated sentient object code does not offer great flexibility in implementation to the developer who is constrained by the set of domain-specific abstractions provided by the programming tool and code generator. However, skilled developers who would typically demand greater implementation flexibility would likely have the ability to customise the generated code.

Maintenance

The maintenance of the code generator lies with a small team who may not be able to timeously and accurately react to user needs for enhancements or changes to be made. The maintenance of the code generator is challenging in itself since the code to generate the sentient object code is complex and dispersed through multiple methods and classes. Since users depend on these external resources for essential maintenance work, the danger is that a lack of responsiveness on their part may lead to abandonment of the programming tool.

Narrow applicability

The code generator is only useful in the domain defined by the sentient object model, and may not easily be extended to other

On balance though, the advantages offered through code generation by the programming tool far outweigh the drawbacks.

5.7 Summary

This chapter described the implementation of a graphical programming tool based on the sentient object model, that significantly reduces the need for developers of mobile, context-aware applications to write syntax. This programming tool fulfills the seventh requirement identified of a programming model for mobile, context-aware applications, namely the provision of an easily accessible and usable development environment. The tool exposes the sentient object model and its fulfillment of the first six requirements of the programming model, in an intuitive and accessible manner to the developer. It is envisioned that by easing the development process, a host of applications will emerge, aiding in the realisation of truly pervasive computing.

The next chapter describes an evaluation of the sentient object model and associated programming tool, based on its application to representative scenarios.

Chapter 6

Applications and Evaluation

The evaluation of Ubiquitous Computing systems is not yet fully understood – Albrecht Schmidt [Alb02]

Applications are the mechanism through which we can test the principles underlying sentient computing – Andy Hopper [Hop00]

Applications are of course the whole point of ubiquitous computing – Mark Weiser [Wei93]

As the quotes above taken from authorities in the area demonstrate, that due to the relative novelty of the research field, there does not exist a common set of criteria by which ubiquitous computing systems are evaluated, but it is generally accepted that systems may be validated through their application. As a direct result, the evaluation of ubiquitous computing systems in general, and context-aware applications in particular, is generally of a qualitative, rather than quantitative, nature. Quantitative measurements of the constituent components of the overall system, using commonly accepted criteria are valuable to an extent, however in this thesis we take a holistic approach to evaluation of the programming model, based on application.

Our evaluation therefore takes the form of qualitatively testing the effectiveness of the sentient object model to adequately model and implement a system, when applied to a representative set of application scenarios. The model is applied to the prototypical imple-

mentation of such applications and a qualitative evaluation made as to the value of the model in the development of context-aware applications.

6.1 Sentient psychiatric couch

The sentient psychiatric couch application aims to evaluate the value of the sentient object model in supporting the development of a context-aware couch that uses sensed weight to recognise who is sitting on it, and in what position. The evaluation is performed with respect to a previous, ad hoc implementation of the application providing similar functionality.

The application is based on an old psychoanalyst's couch that has been installed in the Distributed Systems Laboratory at Trinity College. As a vital tool in psychoanalysis, the couch once served to advance the treatment process by providing patients with an opportunity to relax, undistracted by the visible presence of a therapist, and comfortably report feelings and emotions. In addition, the couch served to define the interaction between therapist and patient as being distinct from ordinary social conversation, and as such facilitated useful communication between therapist and patient.

Now located in an active workplace, the couch serves a different purpose and provides an extremely interesting test case for the sentient object model. Part of the laboratory for approximately two years, the couch no longer serves a psychoanalytic role, but provides an area where members of the laboratory meet and interact with others. In an environment where most people work within semi-private laboratory cubicles, often wearing personal headsets, the couch provides a central focal point where people often come to sit or lie and converse with others. Indeed the couch now forms the main meeting point of the laboratory where issues are discussed and debated, as well as providing a comfortable place to lie down and relax for short periods of time.

The couch has recently been augmented with a set of industrial load sensors placed beneath each leg, enabling the mass, or load, currently on the couch to be measured. Load sensing has a number of properties that are well suited to context acquisition in sentient computing environments [SSL+02], namely:

1. Gravity applies to all physical objects, giving them weight detectable by load sensors.
2. Changes in weight distribution in everyday settings are closely related to interaction in a physical environment.
3. Load sensing technology is mature, inexpensive and unobtrusive.
4. Load sensing is suited to an event-based communication model in which significant changes in detected weight are reported as events.

Since the couch is predominantly used as a seating or lying surface, measured load and its position on the couch are the two major pieces of context information that may be obtained from the sensors, and using this context, the application may perform useful actuation.

The augmented couch infrastructure provides an ideal test-case for our model for a number of reasons. Firstly, the couch and associated infrastructure are already accepted and used as part of the laboratory and therefore the application will not require the introduction of any unfamiliar technology into the environment. This is aligned with Weiser's vision of technology receding into the background in ubiquitous computing environments [Wei91]. Secondly, the couch provides an example of a physical object whose context of use changes frequently throughout the day. Depending on sensed weight and position, the couch may be considered to be in a particular context, with a set of associated actions applicable. The location of the couch within a busy laboratory provides live data to test the system, without the need for extensive simulation. This increases the accuracy of the resulting application. Finally, the couch is able to provide experience of developing and deploying a ubiquitous computing application in a live environment.

6.1.1 System overview

The aim of the sentient psychiatric couch application is to use context information, extracted from sensors, to autonomously control a set of actions. Broadly speaking, the highest level of context information that may be deduced by the couch application, is whether the couch is in use or not. Since we restrict our interest to people, this translates to whether or not somebody

is present on the couch. More refined context information that may be determined is the identity of the 'user' of the couch, and their current location on the couch. In the prototypical implementation of the system the aim is to recognise who is on the couch, what their current position is, and issue a personalised greeting, parameterized with the identity of the user, and their position on the couch. If the user is not recognised, the system should prompt for them to register. This functionality is roughly equivalent to a previous implementation of the system, with the exception that the position of the user was determined, but not reported in the previous implementation.

Since the stated aim of the sentient object model is to ease the development of context-aware applications, the application will be evaluated with regard to this aim.

Previous implementation

One of the attractions of the psychiatric couch as an evaluation scenario for the sentient object model is that a software system has previously been developed and deployed on the couch [Wol03]. This system provided functionality to identify when someone sat on the couch, and to issue a personalised greeting if the person had previously registered with the system. If the person sitting on the couch was not yet registered, the system offered the opportunity to do so.

Whilst this system is perfectly adequate and provides the desired functionality, it was developed in a completely ad hoc and unstructured manner, in common with the majority of existing context-aware applications. The previous implementation saw the hardware tightly integrated into the application, with the result that evolving the hardware without changing the application logic was impossible. Evolving the application logic was also extremely difficult, as there was no separation between context acquisition, fusion and inference, with the logic being incorporated directly into the application code.

Adding or removing hardware from the application would have resulted in significant changes having to be made to the application code and this was not easily achievable by developers who had not been involved in the original implementation.

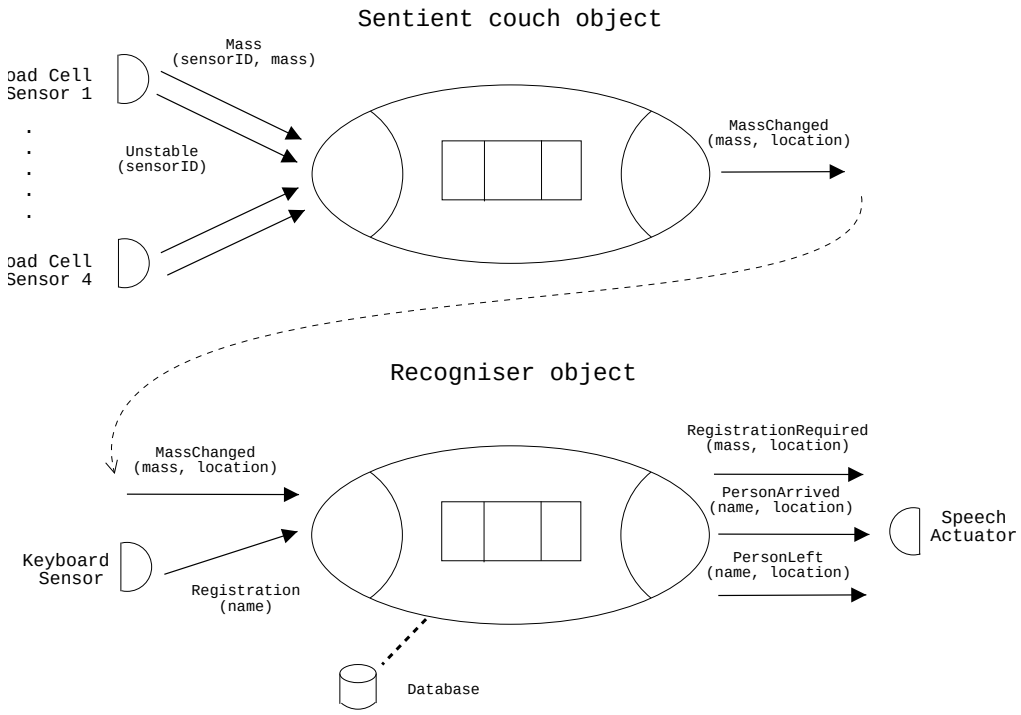

Fig. 6.1: Sentient couch system design

6.1.2 System design

The sentient couch application is designed following the sentient object model and the design process consists of identifying the constituent sensors, objects and actuators of the system, as well as their event interfaces. Two types of sensor, with five sensors in total, two sentient objects, and one type of actuator have been identified and these components and their relationships are illustrated in Figure 6.1. The implementation of each of these components is discussed in detail in the following sections.

6.1.3 Sensors

Recall from Section 4.3, that a sensor in the sentient object model is defined as

> *an entity that produces software events in reaction to a stimulus detected by some real-world hardware device*

Byte no.	Name	Value
1,2	Status 1	OL Overload ST Stable US Unstable
3	-	2C (”,”)
4,5	Status 2	NT Net weight GS Gross weight
6	-	2C (”,”)
7	Polarity	+ Positive - Negative
8-14	Mass data	Actual mass measured at sensor
15,16	Unit	KG Kilograms T Tons
17,18	Control	CRLF

Table 6.1: Format of PT650D reading

In the case of the couch we identify two main categories of sensor. Firstly, the four load cell sensors fitted to the legs of the couch and connected to the weighing indicators detect a real world stimulus that is a change in the weight on the couch. The second category of sensor identified in the application is a keyboard sensor that produces software events in response to keyboard input, following a registration request to a user by the system.

The hardware incorporated in the couch is described in Appendix B and software abstractions of these categories of real world sensors were developed as the sensor components of the application.

Load cell sensor

The weighing indicator outputs a continuous stream of readings, the format of which is illustrated in Table 6.1. The load cell sensor is a software component that uses the standard Java Communications API[1] to communicate with serial ports to which the weighing indicators are attached. An example 18-byte reading from a weighing indicator is illustrated in Table 6.2.

[1] http://java.sun.com/products/javacomm/

O	L	,	N	T	,	-	1	2	3	4	.	5	6	K	G	CR	LF

Table 6.2: Example 18 byte PT650D reading indicating a mass of 1234.56 kg (overload)

Event type	Parameter 0	Parameter 1
Mass	`S_INT` sensorID	`S_DBL` mass
Unstable	`S_INT` sensorID	

Table 6.3: Event types produced by `LoadCellSensor`

There are 4 load cell sensors, one representing each transducer and these read the signal from a weighing indicator, and publish an event containing the mass measured in a stable reading, or an event representing an unstable reading. The two types of events that can be published by the sensor are illustrated in Table 6.3. Events of type **Mass** are published in response to a reading from the weighing indicator with the first 2 bytes set to 'ST', and contain a parameter representing the identity of the transducer, and a parameter containing the mass measurement of the reading. An **Unstable** event is published on reception of a reading from the weighing indicator with the first 2 bytes set to 'US', and contain a single parameter representing the identity of the transducer.

In addition to parsing the readings from the weighing indicator and producing software events, the load cell sensor performs a simple smoothing operation on the mass readings by only publishing a **Mass** event when a specified number of readings, falling within an acceptable range, have been processed. The smoothing function is illustrated in Listing 6.1, where a **Mass** event is only published after 5 readings within 10 kilograms of the last published reading, have been received.

```
if ((mass >= (previousMass - 10)) && (mass <= (previousMass + 10))) {
  readings++;
}else{
  readings = 0;
}
if (readings == 5) {
  raiseMassEvent(mass);
  previousMass = mass;
}
```

Listing 6.1: Smoothing function in load cell sensor

Since the couch is a large, heavy artifact, it is not at all mobile and occupies a fixed location in the laboratory. As such, the load cell sensors instantiate the STEAM location service with a fixed location parameter, and proximity-based filtering is not currently employed in the application, although the proximity of mobile application components to the couch may be incorporated in the design in the future.

```
<?xml version="1.0" encoding="us-ascii" ?>
<!DOCTYPE sensor SYSTEM "sensordefinition.dtd">
<sensor>
 <sensor-name>LoadCellSensor1</sensor-name>
 <sensor-class>ie.tcd.sentient.sensor.LoadCellSensor</sensor-class>
 <event>
  <event-type>Mass</event-type>
  <parameter>
   <param-name>sensorID</param-name>
   <param-type>S_INT</param-type>
  </parameter>
  <parameter>
   <param-name>mass</param-name>
   <param-type>S_DBL</param-type>
  </parameter>
 </event>
 <event>
  <event-type>Unstable</event-type>
  <parameter>
   <param-name>sensorID</param-name>
   <param-type>S_INT</param-type>
  </parameter>
 </event>
</sensor>
```

Listing 6.2: XML descriptor for a load cell sensor

All sensors in the sentient object model have an associated XML descriptor file which describes the interfaces of the sensor, i.e., it describes the types of events published by the sensor, and their associated parameters. The descriptor is developed using the visual programming tool and the descriptor for the load cell sensor is illustrated in Listing 6.2.

Keyboard sensor

The function of the keyboard sensor is to convert information entered at a GUI into a software event that it subsequently publishes. The keyboard sensor publishes an event in response to a prompt to enter a new user's name issued after an unregistered person is detected on the

Event type	Parameter 0
Registration	S_STR name

Table 6.4: Event type produced by `KeyboardSensor`

couch. The sensor produces one type of event, namely a **Registration** event, which has one parameter, the name of the user, as illustrated in Table 6.4.

The keyboard sensor does not perform any processing, but simply a no-op transformation between a name entered in a text field, and a software event. The weight and location of the user are part of the context of the Recogniser sentient object, discussed in Section 6.1.6.

6.1.4 Actuators

The sentient object model defines an actuator as

> *an entity that consumes software events, and reacts by attempting to change the state of the real world in some way via some hardware device*

One type of actuator has been identified in the sentient couch application, this being an actuator that produces an audio stream via a set of speakers, in response to the consumption of an event.

Speech actuator

The function of the speech actuator is to consume the set of event types illustrated in Table 6.5, and perform a transformation to synthesised speech, using a text-to-speech application available on Debian. A **PersonArrived** event is generated by the Recogniser sentient object when a known person begins to use the couch, whilst a **PersonLeft** event is produced by the Recogniser when a person who had been using the couch, leaves. Finally, a **RegistrationRequired** event is produced when an unknown person begins to use the couch.

6.1.5 Couch sentient object

The sentient couch object consumes events from load cell sensors, and performs the main object functions of sensor capture and context acquisition, as well as rule-based inference,

Event type	Parameter 0	Parameter 1
RegistrationRequired	S_DBL mass	S_STR location
PersonArrived	S_STR name	S_STR location
PersonLeft	S_STR name	S_STR location

Table 6.5: Event types consumed by `SpeechActuator`

```
<?xml version="1.0" encoding="us-ascii" ?>
<!DOCTYPE actuator SYSTEM "actuatordefinition.dtd">
<actuator>
 <actuator-name>SpeechActuator</actuator-name>
 <actuator-class>ie.tcd.sentient.actuator.SpeechActuator</actuator-class>
 <event>
  <event-type>RegistrationRequired</event-type>
  <parameter>
   <param-name>mass</param-name>
   <param-type>S_DBL</param-type>
  </parameter>
  <parameter>
   <param-name>location</param-name>
   <param-type>S_STR</param-type>
  </parameter>
 </event>
   . . .
```

Listing 6.3: XML descriptor fragment for a speech actuator

before publishing an event representing the mass currently on the couch, and its location. Development of the sentient object was carried out using the graphical programming tool to specify inputs, logic, and outputs.

Data capture and fusion

The data capture component of the couch sentient object is responsible for receiving the individual events consumed by the object, and storing the fragments of context data contained within them, as well as fusing individual fragments of data to determine higher level context. The first step in object development involved selecting the sensors in which the object is interested from a library containing a set of XML sensor descriptors as illustrated in Listing 6.2. In the sentient couch object, descriptors representing each of the four load cell sensors are selected as producing events of interest to the object. Two forms of sensor fusion are employed in the sentient couch object

```
public void deliver(SC_dsEvent ei) {
  int sensorID_intVal = ei.parValINT(0);
  loadCellSensor1_MassEvent_Bean.setsensorID(sensorID_intVal);
  fusionNetwork.receiveEvidence("sensorID_intVal",""+sensorID_intVal);
  double mass_dblVal = ei.parValDBL(1);
  loadCellSensor1_MassEvent_Bean.setmass(mass_dblVal);
  fusionNetwork.receiveEvidence("mass_dblVal",""+mass_dblVal);
}
```

Listing 6.4: Event delivery updates a Java Bean representing a shadow fact

1. **Sum and average** - the context of the couch is determined by the weight measured at each of the four load cell sensors, the total weight measured on the couch as a whole, and the average weight measured on the couch. The sensor capture component updates individual sensor masses, as well as total and average mass according to the following formulae

$$Mass_{total} = \sum_{n=1}^{4} Mass_{LoadCellSensor_n} \tag{6.1}$$

$$Mass_{average} = \frac{Mass_{total}}{4} \tag{6.2}$$

2. **Bayesian networks** - are used within the sensor fusion component to manage uncertain sensor data. These networks are specified on a per-context basis during object development and are discussed in the next section.

The sensory capture and fusion component stores fragments of context data as a set of facts in the inference engine component. The component dynamically updates these facts as new sensor data is received, using the *shadow fact* mechanism provided by JESS and described in Section 5.5.2. The delivery of events from the load cell sensors causes the update of shadow facts within the inference engine, via corresponding Java Bean objects. The update of the Java Bean representing a **Mass** event from load cell sensor 1 is shown in Listing 6.4, that is an extract from the sentient object code generated by the programming tool.

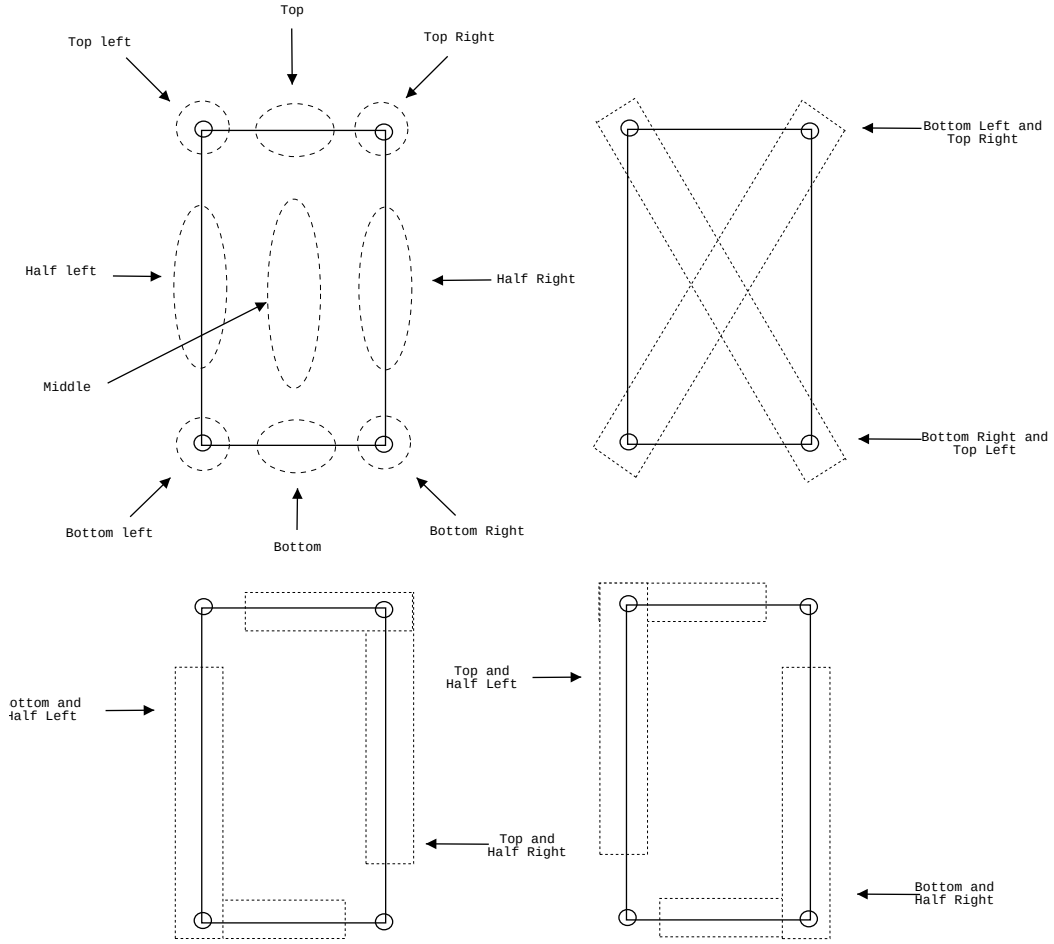

Fig. 6.2: Possible weight distributions on the couch

Context hierarchy

As defined in Chapter 4, the context of a sentient object is represented as a 3-level hierarchy of all possible situations in which the object may be. This representation eases the complexity of developing a sentient object by decomposing the life-cycle of the object into a set of contexts, within which only a subset of all available sensor inputs are used. In the sentient couch application, the two major contexts were identified as **Empty**, when there is no-one on the couch, and **In Use** when there is someone on the couch. Fifteen distinct sub contexts were identified as being possible beneath the **In Use** major context. These fifteen sub contexts

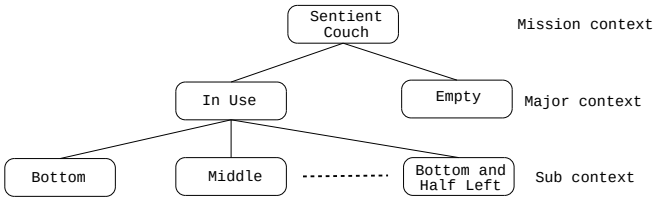

Fig. 6.3: Context hierarchy for the sentient couch

arise from the fact that in order to determine the current context, a comparison is made between the mass of each individual load cell sensor, $Mass_{LoadCellSensor_n}$ and the average weight on the couch, $Mass_{average}$. Each individual sensor may be in one of two states, $Mass_{LoadCellSensor_n} \geq Mass_{average}$, or $Mass_{LoadCellSensor_n} < Mass_{average}$. Since there are four individual sensors, this gives rise to $4^2 = 16$ possible contexts. One of these contexts is captured by the major context **Empty**, leaving 15 sub-contexts, as illustrated in Figure 6.2. The context hierarchy of the sentient couch is illustrated in Figure 6.3, showing only a representative sample of sub contexts for simplicity, whilst Figure 6.4 illustrates how load sensors are used to discriminate between individual contexts.

A *context* in the sentient object model is defined by (1) the set of event types relevant in the context; (2) a set of filters defined over the event types; (3) a probabilistic sensor fusion network; (4) the set of rules active in the context; (5) the set of transition contexts; and (6) the set of output events that may be produced when in the context.

Part of the sensor fusion network for the major context **In Use** is illustrated in Figure 6.5. This network captures prior probabilities, gathered from experimental observation, that a particular sub context is active (e.g., Bottom, Bottom left or Bottom right), given current evidence from the sensors. Evidence from all four load cell sensors is used to determine the probability that the weight is located in a particular position, given historical observations. The experiments conducted to gather evidence for the network consisted of the following steps

- For a known mass $Mass$ at a known position on the couch, $Position_n$, for each load cell sensor LCS_n, measure the probability that the mass measured by each load cell

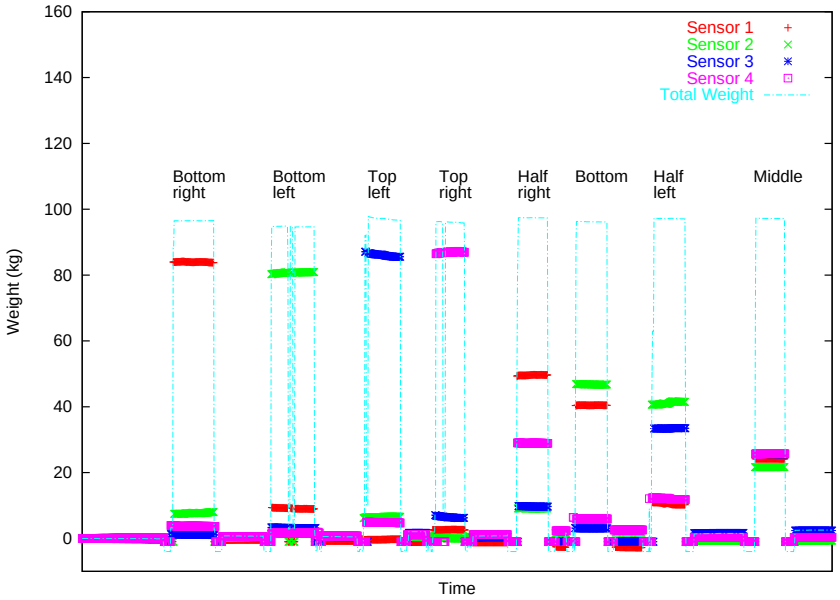

Fig. 6.4: Context determination based on load sensing, showing a subset of potential contexts

sensor $LCS_{n_{Mass}}$ is greater than the average mass on the couch $Mass_{Avg}$

- Iterate, varying mass and position
- Calculate the overall probability for each possible position on the couch $Position_n$, that the mass measured by each of the load cell sensors is greater than the average mass $LCS_{n_{Mass}} > Mass_{Avg}$

These measurements allowed the calculation of the probability that a sensor measures a mass greater than or equal to the average mass, given that a person is located in a particular position on the couch. The prior probabilities are calculated off-line and are used to define the fusion network using the graphical network builder component of the programming tool. The tool then automatically generates a Java class representing the network.

Inference engine

The inference engine component of the couch sentient object maintains a knowledge base containing the set *rules* and working memory containing the set of *facts* that together rep-

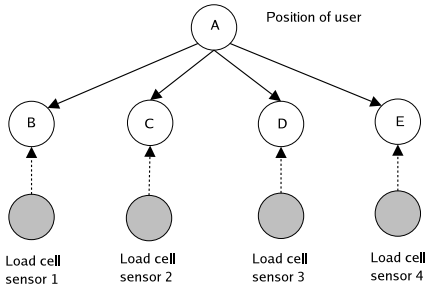

Fig. 6.5: Sensor fusion network for major context **In Use**

	Position		
Load cell sensor 1	Bottom	Bottom left	Bottom right
Above average	0.7	0.9	0.1
Below average	0.3	0.1	0.9

Table 6.6: Conditional probability table for Node B in Figure 6.5

resent the context hierarchy, and govern the behaviour of the sentient object. Two types of facts exist in the working memory of the inference engine

1. **Shadow facts** - each shadow fact represents an event in which the couch is interested. There are eight shadow facts, one for each of the four load cell sensor's two event types, and these are dynamically updated upon delivery of an event. Shadow fact representations are automatically generated by the programming tool for all events consumed by the object.

2. **Unordered facts** - are elements in the working memory of the inference engine that store calculated values such as total and average mass. These facts are updated by rules in the inference engine and are typically hand-coded by the developer to support application logic such as fusion rule outputs.

In addition to facts, the knowledge base of the inference engine contains a set of rules. Three types of rules exist in the inference engine of the couch sentient object

1. **Behavioural rules**

   Behavioural rules are rules that govern the behaviour of the object and of which a subset

```
(defrule InUse-Bottom
 (active-major-context InUse)
 ?cu <- (context-updated true)
 ?aw <- (average-weight (averageweight ?X))
 (SmartCouch_LoadCellSensor1_MassEvent_Bean (sensorID 1)(mass ?LoadCellSensor1_MassEvent))
 (test (> ?LoadCellSensor1_MassEvent ?X))
 (SmartCouch_LoadCellSensor2_MassEvent_Bean (sensorID 2)(mass ?LoadCellSensor2_MassEvent))
 (test (> ?LoadCellSensor2_MassEvent ?X))
 (SmartCouch_LoadCellSensor3_MassEvent_Bean (sensorID 3)(mass ?LoadCellSensor3_MassEvent))
 (test (< ?LoadCellSensor3_MassEvent ?X))
 (SmartCouch_LoadCellSensor4_MassEvent_Bean (sensorID 4)(mass ?LoadCellSensor4_MassEvent))
 (test (< ?LoadCellSensor4_MassEvent ?X))
 =>
 (retract ?cu)
 (assert (active-sub-context Bottom))
)
```

Listing 6.6: Generated transition rule for the transition **In Use-Bottom**

are valid in each particular context. At the lowest level, the behaviour of a sentient object consists of the publication of events, as caused by the evaluation of rules, for consumption by relevant actuators.

The couch sentient object publishes events of type **MassChanged** (see Table 6.7) when its context, that is the mass and its location on the couch, changes. The behavioural rules of the couch are thus a set of rules that check whether a particular sub context is active in their LHS, and if it is, publish an appropriate event on the RHS of the rule. An example of a behavioural rule composed in the graphical rule-builder and generated for the couch is illustrated in Listing 6.5 for the sub context **Bottom**.

```
(defrule Bottom
 (active-sub-context Bottom)
 =>
 (bind ?MassChangedEvent (new java.util.Vector))
 (bind ?param0 (new ParameterInstance "mass" "S_DBL" (call String valueOf ?*weight*)))
 (call ?MassChangedEvent add ?param0)
 (bind ?param1 (new ParameterInstance "location" "S_STR" "Bottom"))
 (call ?MassChangedEvent add ?param1)
 (call ?*producer* produceEvent MassChanged ?MassChangedEvent)
)
```

Listing 6.5: Generated behavioural rule for sub context **Bottom**

2. **Transition rules**

   Transition rules serve to identify when transition to another context is indicated, and perform this transition. Transition rules are associated with particular contexts and are only eligible to fire when the context to which they relate is active. In addition to the context being active, the conditions under which the transition occurs must be met. An example transition rule generated for the couch object is illustrated in Listing 6.6.

```
(defrule fusion
 ?aw <- (average-mass (averagemass ?X))
 (SmartCouch_LoadCellSensor1_MassEvent_Bean (mass ?mass1))
 (SmartCouch_LoadCellSensor2_MassEvent_Bean (mass ?mass2))
 (SmartCouch_LoadCellSensor3_MassEvent_Bean (mass ?mass3))
 (SmartCouch_LoadCellSensor4_MassEvent_Bean (mass ?mass4))
 =>
 (call ?*contextbean* contextDataReceived)
 (bind ?*totalmass* (+ ?mass1 ?mass2 ?mass3 ?mass4))
 (bind ?*averagemass* (/ ?*totalmass* 4))
 (if (<> ?*averagemass* ?X) then
 (modify ?aw (averagemass ?*averagemass*)))
)
```

Listing 6.7: Fusion rule calculating total and average mass on couch

Event type	Parameter 0	Parameter 1
MassChanged	`S_DBL` mass	`S_STR` location

Table 6.7: Event type produced by `SentientCouch` object

This rule states that if the current active major context is **In Use**, if the context information has recently been updated, and if load cell sensor 1 and 2 are measuring a mass greater than the average mass on the couch, whilst load cell sensors 3 and 4 are measuring a mass less than the average, then the sub context **Bottom** is activated.

3. **Custom rules**

   Whilst behavioural rules and transition rules for the sentient couch object were easily encoded using the graphical rule builder incorporated into the programming tool, there was an additional requirement in the object for application-specific rules, specifically rules performing sensor fusion, and these were developed by hand. Whilst it is desirable to abstract away from the implementation of rules as much as possible, some application-specific rules require hand coding, but the programming tool significantly eases the development of these rules by providing a centralised rule repository in the `controller.clp` file. An example of an application-specific fusion rule that calculates the average mass on the couch, is illustrated in Listing 6.7.

Output events

The sentient couch object publishes events of type **MassChanged**, containing parameters representing the total mass on the couch, and its position, as illustrated in Table 6.7.

Field	Type
user_name	varchar
attribute_name	varchar
value	varchar

Table 6.8: Schema of the `Couch` table

6.1.6 Recogniser sentient object

During analysis of the sentient couch system, two sentient objects were designed. Whilst the sentient couch produces events that represent the current mass on the couch, and its position, a *recogniser* sentient object uses this information to identify the user of the couch, as well as register new users with the system.

Data capture and fusion

The recogniser object consumes two types of events, namely **MassChanged** events produced by the couch object, and **Registration** events produced by a keyboard sensor. There is no need to perform any sensor fusion in the recogniser object and the sensory capture and fusion component thus serves to extract and store context data from events consumed by the object.

Context data is stored as facts in the inference engine and in the case of the recogniser object, the context data consists of the mass currently on the couch, and the location of the mass on the couch, as well as the name of the user. The mass and location are extracted from **MassChanged** events produced by the sentient couch object, and are stored in the inference engine as a shadow fact, updated each time a new event is consumed.

The name of the user is inferred from the mass by performing a lookup on a database table that stores the relation between mass and user name. The table has 3 fields as illustrated in Table 6.8. The `attribute_name` field has the value 'Mass' for all records in the table.

Context hierarchy

The recogniser object can exist in one of three major contexts, either there is a known user on the couch, an unknown user on the couch, or no user on the couch. In the context of there being a known user on the couch, there is one associated sub context that is the issuing

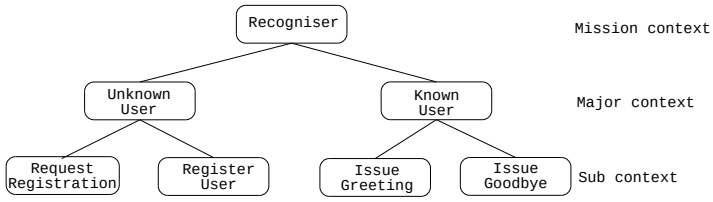

Fig. 6.6: Context hierarchy for the recogniser sentient object

of a personalised greeting to the user, welcoming them to the location in which they are positioned on the couch.

In the context of the user being unknown, there are two associated sub contexts, namely the issuing of a request to the user to register with the system, and performing the registration of the new user. The context hierarchy is illustrated in Figure 6.6.

Inference engine

In addition to a set of shadow and unordered facts in working memory, the knowledge base of the recogniser object's inference engine component contains a set of behavioural and transition rules.

1. **Behavioural rules**

   Behavioural rules encode behaviours of the object, which in the case of the recogniser object is the production of events to be consumed by a speech actuator.

   An example of a behavioural rule defined in the recogniser object is illustrated in Listing 6.8. The rule states that when the object is in the sub context **Request Registration**, a **RegistrationRequired** event is produced, containing the mass and location of the user.

```
(defrule RequestRegistration
 ?ac <- (active-sub-context RequestRegistration)
 =>
 (retract ?ac)
 (bind ?SpeechActuatorEvent (new java.util.Vector))
 (bind ?param0 (new ParameterInstance "mass" "S_DBL" (str-cat ?*mass*)))
 (call ?SpeechActuatorEvent add ?param0)
 (bind ?param1 (new ParameterInstance "location" "S_STR" ?*location*))
 (call ?SpeechActuatorEvent add ?param1)
 (call ?*producer* produceEvent RegistrationRequired ?SpeechActuatorEvent)
)
```

Listing 6.8: Generated behavioural rule in sub context **Request Registration**

2. **Transition rules**

   Transition rules govern the transition between active contexts and an example transition rule in the recogniser object is illustrated in Listing 6.9. This rule states that if the object is in the major context **Unknown Person**, and the user has not yet registered, then the sub context **Request Registration** should be activated.

```
(defrule UnknownPerson-RequestRegistration
 (active-major-context UnknownPerson)
 (hasRegistered false)
 =>
 (assert (active-sub-context RequestRegistration))
)
```

Listing 6.9: Transition rule for the transition **Unknown Person-Request Registration**

3. **Custom rules**

   A custom rule is developed within the rule-base to perform the database lookup required by the recogniser object. A Java class which performs database access via JDBC was easily incorporated and accessed using JESS syntax within the rule-base.

Output events

The recogniser object produces three types of events, as consumed by the speech actuator, and illustrated in Table 6.5.

6.1.7 Extending the application

The sentient couch system implemented according to the sentient object model and using the programming tool, exhibits the same functionality as the previous unstructured implementation. Once this minimal functionality had been demonstrated, we planned to extend the sentient couch by adding an additional sensor and actuator to the system.

Extending the previous, unstructured implementation to encompass new, or different, sensors or actuators, or altering the application logic to exhibit different behaviours is a complex and very difficult task due to the close coupling between context acquisition and use within the application. Extending the functionality of the application was only accessible to experienced C++ developers.

To extend the functionality of the sentient couch application implemented using the graphical programming tool, we added an additional identity sensor in the form of a barcode sensor, and an additional actuator, in the form of an electronic mail notification actuator. The barcode sensor and electronic mail actuator components are described in detail in Appendix B.

To incorporate these additional components, the following steps were performed with the programming tool;

1. Load the sentient couch object descriptor, generated when developing the original application.
2. Add the barcode sensor and e-mail actuator descriptors to the object specification.
3. Specify additional rules based on the new input and output events available to the object. Additional contexts may also be developed.
4. Generate and compile the sentient object code.
5. Add to, or edit the rule-base in the `controller.clp` file, run, and test. Further changes to the rule-base do not require re-compilation.

6.1.8 Comparison with existing implementation

Implementation of the couch application following the sentient object model yielded some immediate advantages over the previous implementation.

Code size

Importantly, in the application created using the programming tool, all procedural syntax was automatically generated and the developer was only required to write a small number of custom declarative rules by hand. Whilst the size of the code generated by the programming tool (400 KB) was approximately 45% smaller than the previously implemented application (581 KB), the generated code makes use of external libraries of approximately 900 KB in size. Although these libraries represent a significant overhead, this overhead remains negligible in

terms of modern mobile storage, which is now often measured in hundreds of megabytes, and increasingly, gigabytes[2]. For example, the iPAQ mobile computer used in other experiments provided 128 MB of RAM.

Decoupling of components

Following the approach of the sentient object model, it was possible to decouple application development into the development of sensors, actuators, and sentient objects, and have different developers working on the project simultaneously. All that needed to be communicated between the developers was the event interface of each component, in the form of an XML descriptor. This parallelisation significantly speeded up the development process.

Extensibility

Extending the 'vanilla' implementation of the sentient couch by adding additional components in the form of a barcode sensor and an electronic-mail-based notification actuator, was greatly eased by the programming tool through the ability to load and edit object specifications, and rapidly regenerate the code implementing the object.

Maintenance

The centralisation of application logic in a single rule-base file per sentient object considerably simplified the maintenance and evolution of the application. In the previous implementation of the system, application logic was dispersed throughout multiple files in the application, complicating maintenance and evolution.

6.1.9 Development challenges

The process of implementing the sentient couch application highlighted a number of development challenges which can be abstracted to ubiquitous application development in general.

[2]In 2004, one gigabyte of compact flash memory is available for under USD100

A 'living' environment

Since the hardware on which the system is based was already installed in a relatively compact work environment used by up to 15 people on a daily basis, the interaction of people with the system could not be ignored during development and testing. Since the couch has traditionally provided a place to meet and relax, people continued to use it as such with the result that stable configurations required for testing the system could not always be attained during normal working hours. Indeed, evidence of work on the couch often attracted extra attention to the hardware which served to further complicate testing.

Privacy

In order to recognise users on the couch at different points in time, the system needs to store the relation between a name and a weight to persistent storage. Weight is particularly sensitive information in some western European cultures, and is typically not information people wish to reveal about themselves. Realisation that the system stores an association between an accurate weight measurement and an individual's name, put a number of people off using the system.

A potential solution to this privacy issue is to encrypt the weight value with a one-way hash function (such as MD5), before storing it in the database. The problem with this approach is that in order to recognise a user, the measured weight would have to be exactly the same each time, since it is only possible to calculate equality of hash values, and the sensors will seldom provide the same measurement at different times. A symmetric encryption algorithm solves this problem and provides sufficient privacy.

6.1.10 Perspective

The sentient couch application provided a valuable opportunity to make a direct comparison between an ad hoc implementation of a context-aware application, and implementation of the same application using the programming model described within this thesis. Whilst it was not possible to accurately measure the historical development effort expended on the ad hoc implementation, the use of the programming model obviated the need for the

sentient object developer to write any low-level, procedural code. The only syntax that was required to be developed by hand was the implementation of a small amount of application specific functionality (in this case, a database lookup via JDBC), and the incorporation of this functionality into the generated application. Incorporation of additional functionality into the application required the editing of only one file, with context capture, representation, and fusion handled by the automatically generated, low-level code.

6.2 Sentient vehicle

The sentient vehicle application applies the sentient object paradigm to a model car augmented with sensors, in order to enable it to sense its environment, and drive autonomously, whilst potentially co-operating with other vehicles. The sentient vehicle application exposes important characteristics to be considered in relation to the development of mobile, context-aware applications that were not raised in the development of the sentient couch. These include ad hoc wireless communication in an infrastructureless environment, as well as location-awareness and associated proximity-based filtering of communication.

6.2.1 System overview

The aim of the sentient vehicle application is to apply the sentient object model and associated programming tool to enable a model vehicle to operate autonomously, based on context data acquired from sensors, and using actuators. In terms of a sentient vehicle, the most important context data permitting autonomous operation is the current location and orientation of the vehicle, whether an obstacle is present in the path of the vehicle, as well the status of traffic signals in the vehicle's path. Given this context data obtained from a set of sensors on the vehicle, there are a number of distinct scenarios that a sentient vehicle may operate in.

Forward obstacle detection

This scenario simply involves the vehicle travelling in a straight line and detecting obstacles within its path using forward-facing ultrasonic range finders. If an obstacle is detected within

a predetermined critical range by any of the sensors, the vehicle adopts the safe behaviour of coming to a halt.

Forward obstacle avoidance

As an extension to the scenario in which the vehicle simply comes to a halt following detection of an obstacle, in this scenario, the vehicle uses data fused from three forward-facing ultrasonic range finders to determine the location of the obstacle, with a bounded probability, and then attempts to avoid the obstacle by turning away from it and continuing to travel forward. This scenario illustrates the fusion of readings from multiple ultrasonic sensors, in order to determine the relative position of an obstacle encountered in the path of the vehicle, and to infer appropriate actions to avoid the obstacle whilst still moving.

Traffic signal obeyance

In this scenario, the vehicle obeys a simulated traffic signal autonomously, by stopping if it is approaching a red signal in close proximity, and proceeding otherwise. This scenario illustrates event-based communication using proximity filtering, as provided by the event service incorporated in the sentient object model. Proximity filters are specified by the event producer, and the traffic light produces events which are valid within a specific area. In this scenario, the traffic light sensor produces status events valid within a circular area of radius 10m. The vehicle is able to filter events so as only to receive events from a traffic light that it is travelling towards, and further more, may specify a filter that ensures only status events requiring some action on the part of the vehicle, are delivered (e.g., a red light).

Autonomous navigation

In this scenario, the vehicle navigates autonomously between a set of predefined waypoints, using data fused from location and compass sensors to determine position and heading. This scenario demonstrates autonomous and meaningful behaviour by the sentient vehicle, in following a specific route, with the vehicle basing its behaviour on context data acquired from sensors.

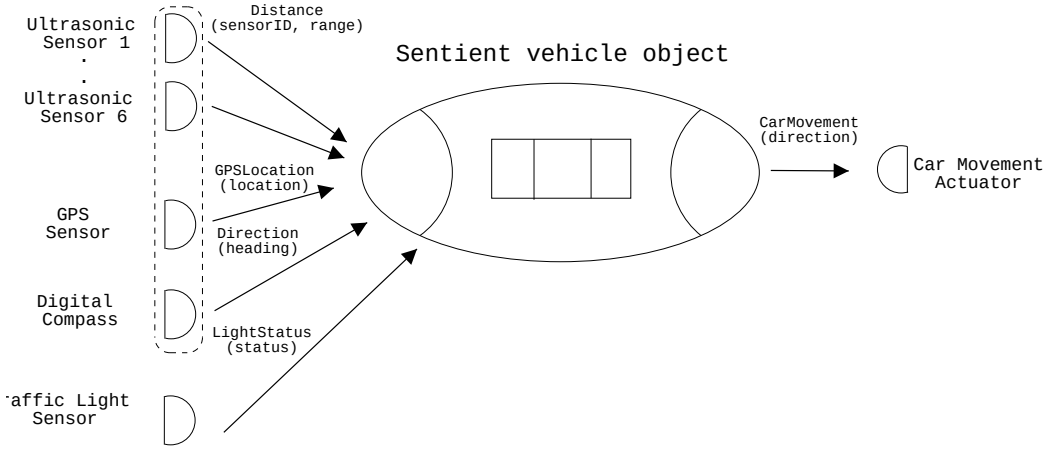

Fig. 6.7: Sentient vehicle system design

The set of scenarios are not mutually exclusive, for instance, a vehicle will obey traffic light signals whilst navigating between waypoints, as well as detecting and avoiding obstacles whilst navigating between waypoints, or obeying traffic signals. Certain scenarios have a higher *priority* than others, for instance, obstacle detection and avoidance will always take precedence over all other potential scenarios.

6.2.2 System design

The sentient vehicle design follows the principles embodied in the sentient object model and involves identification of the relevant sensors, actuators, and sentient objects in the system, and their event interfaces. Four types of sensors, one sentient object, and one type of actuator have been identified in the design of the system and these components and their relationships are illustrated in Figure 6.7, whilst Appendix D provides a detailed description of the system hardware.

Event type	Parameter 0	Parameter 1
Distance	`S_INT` sensorID	`S_INT` range

Table 6.9: Event type produced by `DistanceSensor`

6.2.3 Sensors

There are four types of sensor in the sentient vehicle application: distance sensors, a location sensor, an electronic compass, and a simulated traffic light. Each of these sensors and their event interfaces are detailed in the following sections.

Distance sensor

The distance sensor is a software component which converts real-world events (range readings) from the SRF08 ultrasonic rangefinder transducer (see Appendix D), into software events. As such, the sensor consists of software components resident on both an OOPIC microcontroller (described in Appendix D), and an iPAQ handheld computer. On the microcontroller, the registers (see Table D.2) of each of the SRF08 sensors are read via the I2C bus, after a ranging command. Each range value is then communicated to the iPAQ via RS232, where the range reading is converted into an event of type **Distance** as illustrated in Table 6.9, and published over the wireless network interface of the iPAQ. The code on the microcontroller to communicate with the transducers, is written in Java-like syntax, whilst the event publisher code of the sensor component is developed on the iPAQ, using C++ for Windows CE.

Location sensor

The location sensor publishes events containing the current geographical location of the vehicle as a longitude and latitude pair encapsulated in the `S_POS` STEAM event type, as determined by the GPS receiver. The location sensor software component interrogates the STEAM location service running on the iPAQ, for the current location, parsed by the location service from NMEA sentences produced by the GPS receiver in response to real-world events in the form of satellite signals. The sensor produces one type of event, illustrated in Table 6.10. The code for the location sensor was written using C++ for Windows CE [O'C04].

Event type	Parameter 0
GPSLocation	`S_POS` location

Table 6.10: Event type produced by `LocationSensor`

Event type	Parameter 0
LightStatus	`S_STR` status

Table 6.11: Event type produced by `TrafficLightSensor`

Traffic light sensor

The traffic light sensor is a software component which emulates a traffic light by producing events containing the status of a real traffic light. This type of sensor produces one type of event, as illustrated in Table 6.11. The traffic light sensor code was developed in C++ on Windows XP, and simply switches the state of a traffic light in timed phases, periodically (once per second) publishing an event containing the current status.

Heading sensor

The heading sensor receives events from the electronic compass and produces an event containing the current orientation of the vehicle, as illustrated in Table 6.12. The heading sensor consists of a software component resident on the OOPIC microcontroller, written in Java-like syntax to communicate with the transducer via I2C, and a component resident on the iPAQ, written in C++ for Windows CE, and serving to publish event notifications containing the current value of the electronic compass.

6.2.4 Actuators

The sentient vehicle application defines one type of actuator, which sends commands to the servos controlling the car, via the existing RC wireless interface.

Event type	Parameter 0
Heading	`S_INT` heading

Table 6.12: Event type produced by `HeadingSensor`

Event type	Parameter 0
CarMovement	S_STR command

Table 6.13: Event type consumed by `CarMovementActuator`

Vehicle movement actuator

In its original form, the car was controlled via an RC transmitter by which a human operator issues a set of commands to the car (forward, back, left, right) through the closing of electronic circuits by hand. In order to control the car via the microcontroller, these circuits were closed, with relays placed in the circuit that can be controlled by the microcontroller [O'C04]. The original RC control board was removed from the hand-held control unit and integrated into the vehicle itself, connected to the OOPIC microcontroller.

The vehicle movement actuator is a software component, written in C++ for Windows CE, and running on the iPAQ resident on the vehicle. This component consumes software events published by the sentient vehicle object, and transforms them to real world events controlling the vehicle. The actuator converts these software events into commands that are sent to the OOPIC microcontroller via RS232, and in turn are transmitted to the RC control board connected to the microcontroller, by closing the appropriate relays.

6.2.5 Vehicle sentient object

The vehicle sentient object consumes events from distance sensors, a location sensor, traffic light sensors, and an electronic compass sensor, performing the object functions of sensor capture and fusion, context representation, and intelligent inference, before publishing events to control the movements of the vehicle.

The sentient object resides on a laptop-based controller, separate from the vehicle itself. The execution of the sentient object on the laptop was necessitated by the generation of Java code by the programming tool. The Java implementation would not run on the CE operating system, necessitating the use of a Linux-based device. The resulting system configuration is illustrated in Figure 6.8, and results in the sentient object executing on a different host from the sensors and actuators, clearly demonstrating the distribution within the system.

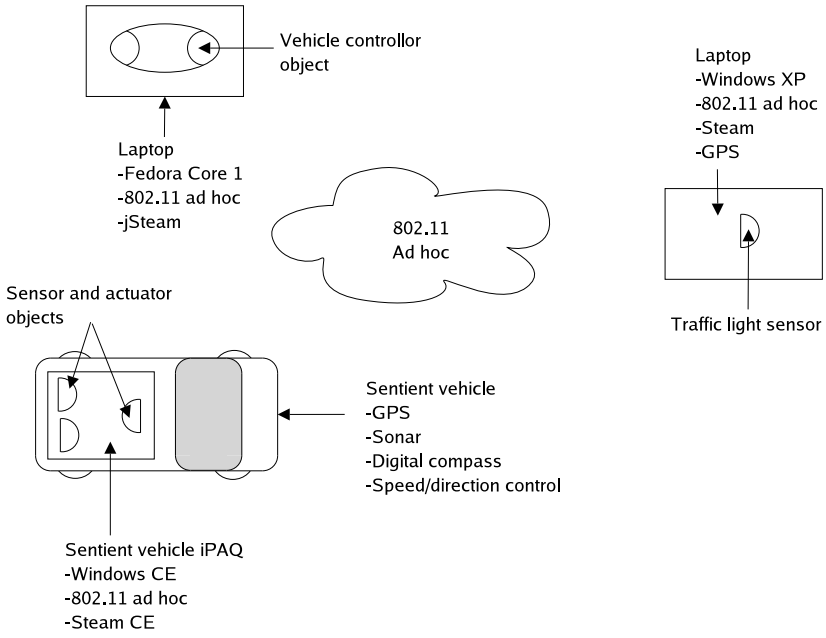

Fig. 6.8: Configuration of the sentient vehicle application

Data capture and fusion

The data capture and fusion component of the vehicle sentient object is responsible for the fusion of individual events received from sensors, as well as the storage of fragments of context data.

Although this component is shown to be conceptually separate in the sentient object model, during practical development of a sentient object using the visual programming tool, the specification of data capture and fusion is performed on a per-context basis, and is part of the context specification as described in Section 5.4.2.

Context hierarchy

The context hierarchy identified for the sentient vehicle application scenarios discussed in Section 6.2.1 is illustrated in Figure 6.9.

The overall mission context of the vehicle is to drive from point A to point B whilst avoiding obstacles in the path of the vehicle, obeying approached traffic signals, and potentially

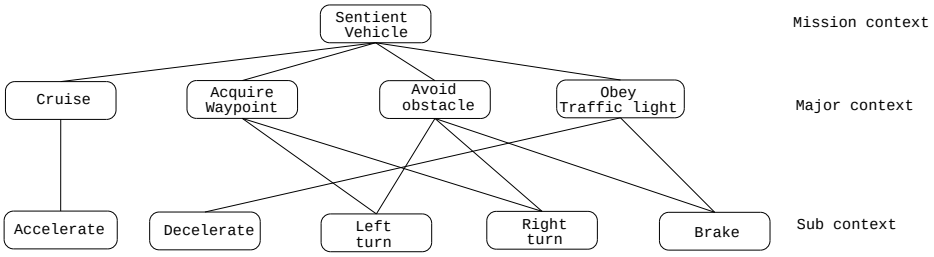

Fig. 6.9: Context hierarchy for vehicle sentient object

following a set of pre-defined waypoints. The constituent major contexts identified are as follows

1. **Cruise** - in this context the vehicle travels in a straight line, accelerating to cruising speed. This context can thus activate the **Accelerate** sub context

2. **Acquire waypoint** - in this context the vehicle turns to acquire the correct bearing to take it towards the next waypoint. This context can activate either the **Left turn** or **Right turn** sub context.

3. **Avoid obstacle** - in this context the vehicle avoids an obstacle by either stopping, or turning away from the obstacle. This context can activate the **Left turn**, **Right turn**, or **Brake** sub context.

4. **Obey traffic light** - in this context the vehicle obeys signals from a traffic light that it is approaching. This context is able to activate the **Decelerate** or **Brake** sub context.

Five sub-contexts were identified with respect to the actions which may be taken by the vehicle, and the relationships between contexts in the hierarchy is illustrated in Figure 6.9.

By way of example, the sensor fusion network defined for the major context **Avoid obstacle** is illustrated in Figure 6.10, and performs probabilistic fusion of readings from the three forward-facing ultrasonic range finders in order to detect obstacles in the path of the vehicle. This context is transitioned to from any other major context, when any of the forward-facing ultrasonic range-finders detect an obstacle closer than a threshold range.

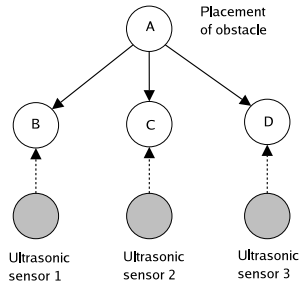

Fig. 6.10: Sensor fusion network for major context **Avoid obstacle**

Obstacle Location	**Sensor 1 (Node B)**		**Sensor 2 (Node C)**		**Sensor 3 (Node D)**	
	Detected	Undetected	Detected	Undetected	Detected	Undetected
Straight ahead	0.2	0.8	0.99	0.01	0.1	0.9
Left	0.05	0.95	0.2	0.8	0.85	0.15
Right	0.95	0.05	0.1	0.9	0.01	0.99

Table 6.14: Conditional probability table for Nodes B, C and D in Figure 6.10

Using prior probabilities calculated for each of the individual sensors (see Appendix D), the Bayesian network fuses the output from each of the three forward facing sensors, and returns the probability that the obstacle is located straight ahead, or to the left or the right of the vehicle's path. The conditional probability table for the hypothesis node A of the fusion network, as calculated from the experiments detailed in Appendix D, is illustrated in Table 6.14. Intuitively, the action of the vehicle will be determined by the relative position of the obstacle, and this was encoded into rules, by the developer. An obstacle detected to the left will cause the vehicle to turn right, and vice versa, whilst an obstacle located straight ahead will cause a random left or right turn, and subsequent re-evaluation of the obstacle position.

Inference engine

The inference engine component of the vehicle sentient object consists of a knowledge base composed of behavioural and transition rules specified by the developer, as well as dynamically updated shadow facts, based on Java beans generated by the programming tool. Examples of rules generated by the programming tool, from visual specification by the developer

```
(defrule Avoid_obstacle-Left_turn
 ?ac <- (active-sub-context-Left_turn)
 =>
 (bind ?CarCommand (new java.util.Vector))
 (bind ?param0 (new ParameterInstance "command" "S_STR" "l"))
 (call ?CarCommand add ?param0)
 (call ?*producer* produceEvent CarMovement ?CarCommand)
 (retract ?ac)
)
```

Listing 6.10: Generated behavioural rule for sub context **Left turn**

```
(defrule Avoid_obstacle-Left_turn
 (SentientVehicle_Obstacle_location_Bean (right ?probability_right))
 (test (> ?probability_right 0.7))
 =>
 (assert (active-sub-context Left_turn))
)
```

Listing 6.11: Generated transition rule for the transition **Avoid obstacle-Left turn**

are illustrated below

1. **Behavioural rules**

   Behavioural rules govern the behaviour of a sentient object through publication of events for consumption by actuators and other objects, and are only valid in specific contexts. The sentient vehicle object publishes events, based on behavioural rules, for consumption by the car movement actuator, which controls the movement of the vehicle. An example of a behavioural rule generated for the sentient vehicle is illustrated in Listing 6.10, for the sub context **Left turn**. This rule publishes a **CarMovement** event with the *command* parameter set to "l" to perform a left turn. The fact that a sub context is only active for a short period of time can be seen in the retraction of the fact that it is active, as soon as the event has been published.

2. **Transition rules**

   An example of a transition rule generated by the programming tool for the transition from major context **Avoid obstacle** to the sub context **Left turn** is illustrated in Listing 6.11. This rule states that when in the active context of **Avoid obstacle**, and the probability that there is an obstacle to the right of the vehicle is greater than 0.7, then transition to the sub context of **Left turn**. The major context remains active.

   Intuitively, certain transitions between contexts have a higher priority than others, for

example, a transition to the major context **Avoid obstacle** should assume priority over any other transition indicated at the same time (for example acquiring a waypoint). The priority of transition rules may be specified by the developer, by defining the *salience* of the rule. Each rule has an associated salience, representing the relative priority of the rule, and activated rules of the highest salience always fire first. The salience of particular transition rules may be specified by editing the generated rule-base, and adding the command `(declare (salience` x`))` where x represents the salience value. The rule governing the transition to the major context **Avoid obstacle** is assigned the highest salience, to ensure it always fires ahead of any other rules.

Custom behaviour

Whilst transition between contexts, and most behaviours are easily specified using the graphical programming tool, application-specific behaviour controlling navigation of the vehicle is developed outside the tool and integrated into the code generated by the programming tool.

The custom behaviour in the sentient vehicle application is concentrated in the **Acquire waypoint** context, and consists of functionality to direct a vehicle towards its next waypoint. This functionality consists of methods to calculate the distance from the current location of the vehicle, to a specified waypoint, as well as calculating the bearing to be followed to get to the waypoint. Since the current bearing of the vehicle is part of the context of the sentient object (updated by the heading sensor), in order to 'acquire' a waypoint, the vehicle turns until its heading is within a defined range (5° in this instance) of the required bearing, as illustrated in Figure 6.11. We make use of great circle navigation equations [Ste00] (see Appendix D) to calculate distance and bearing between waypoints, and this functionality is incorporated into the application code generated by the programming tool.

Incorporating custom behaviour

The rule-base of the application resides in the `controller.clp` file, separate from other application code, and is the only file that needs to be edited by the developer to incorporate the custom navigational behaviour. The file contains the set of generated rules (examples

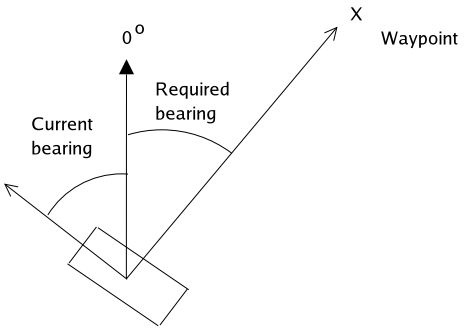

Fig. 6.11: Waypoint acquisition

of which are illustrated in Listings 6.10 and 6.11), and custom behaviour may be added by simply adding rules. The rules are written in JESS, which is tightly linked to Java, so additional functionality may make use of any Java APIs. JESS is programmed declaratively, and custom rules may be added to the rule-base with ease.

The custom waypoint acquisition behaviour was incorporated into the application by firstly implementing great circle distance and bearing equations in a Java class, which was then made available to the application rule-base. Methods in the class are then available to rules that update the bearing and distance to a waypoint, storing this information as facts available to behavioural rules. The update of the distance and bearing to the next waypoint, following a location event received by the sentient object, is illustrated in Listing 6.12. This rule calls methods on the navigation helper class, available via the global variable `?*navigationhelper*`, to calculate bearing and distance.

```
(defrule UpdateDistanceAndBearing
  (SentientCar_LocationSensor_GPSLocation_Bean (position ?position))
  =>
  (bind ?currentlat (call ?position lat))
  (bind ?currentlng (call ?position lng))
  (bind ?waylat (call ?*waypoint* lat))
  (bind ?waylng (call ?*waypoint* lng))
  (bind ?*distance* (call ?*navigationhelper* convertCalculateDistance
                    ?currentlat ?currentlng ?waylat ?waylng))
  (bind ?*bearing* (call ?*navigationhelper* convertCalculateBearing
                   ?currentlat ?currentlng ?waylat ?waylng))
  (bind ?*variance* (- ?*heading* ?*bearing*))
)
```

Listing 6.12: Custom rule written to access a navigation helper class

Behavioural rules could then be written that reason based on the distance and bearing context data, producing appropriate behaviour (i.e., turning the vehicle in the correct

```
/** current is the current bearing, required is the required bearing */
variance = abs(required - current)
if (variance < 180)
  if (required > current)
    turn right
  else
    turn left
else
  if (required > current)
    turn left
  else
    turn right
```

Listing 6.14: Algorithm to determine which direction to turn to acquire waypoint

direction until its current bearing is equal to the required bearing). A transition rule was developed that tests the variance between the current bearing of the vehicle, and the required bearing, and if it is greater than (5°), activates the **Acquire waypoint** context. This rule is illustrated in Listing 6.13.

```
(defrule Cruise-Acquire_Waypoint
?ac <- (active-major-context Cruise)
 (SentientVehicle_LocationSensor_GPSLocation_Bean (position ?position))
 (test (> ?*variance* 5))
 =>
 (retract ?ac)
 (assert (active-major-context Acquire_Waypoint))
)
```

Listing 6.13: Transition rule to acquire a waypoint

Within the **Acquire waypoint** context, a further rule tests to see which way the vehicle should turn in order to acquire the waypoint quickest, based on the difference between the current bearing, and the required bearing, and using the algorithm illustrated in Listing 6.14, where `required` and `current` refer to the bearing required to travel to the waypoint from the current location, and the current bearing of the vehicle, respectively. Based, on this algorithm, either the **Left turn** or **Right turn** sub context is activated.

Output events

The sentient vehicle object produces one type of event as output, as consumed by the car movement actuator, and illustrated in Table 6.13.

6.2.6 Evaluation

Our evaluation of the sentient vehicle application assesses the value of applying the sentient object model to the sentient vehicle application based on a number of criteria, as discussed

below.

Ease of development

The ease of application development unifies other criteria by assessing the relative effort and expertise required to develop the application based on the sentient object model and employing the visual programming tool and associated abstractions.

Ease of development is difficult to measure as well as being highly subjective, however a number of points related to this metric arose from the sentient vehicle application that reinforce the value of the sentient object model in supporting the development of mobile, context-aware applications.

1. **Decoupling of development effort** - the anonymous event-based communication mechanism of the sentient object model allows sensor, actuator, and object components to be developed by separate developers in parallel. This proved extremely useful in the sentient vehicle application, allowing one developer to concentrate on building sensors and actuators to interface with hardware, whilst another developer focused on building the sentient object.

2. **Systematic approach** - the sentient object model provided a systematic, structured approach to scoping inputs and outputs, fusing multi-modal data, and specifying rules.

3. **No need to write low-level sentient object code** - the visual programming tool obviated the need for the sentient object developer to write any low-level Java code. Application logic was maintained in a single file containing the rule-base expressed in high-level, declarative CLIPS syntax.

4. **Debug** - the event-based communication paradigm enabled the developer to debug the application effectively by subscribing to application events. In the sentient vehicle application, an interface (illustrated in Appendix B) was rapidly constructed which presented event streams between sensors, sentient object, and actuators, to the developer, easing the process of debugging the application.

Accessibility

The structured approach of the sentient object model provided a framework with which to approach the design of the application, and consequently significantly simplified the design process. By decomposing the application into sensors, actuators, and a sentient object with associated contexts, rules, and a systematic approach to fusing sensor data, a coherent design was quickly formed.

The graphical programming tool provided an intuitive, cross-platform development environment based on the sentient object model, and made the development of applications based on sentient objects eminently accessible, through significantly reducing the need to write low-level syntax.

Extensibility

The sentient object model contributed significantly to the extensibility of the vehicle application through both the event-based communication mechanism and the code generation capabilities of the visual development tool.

The event-based communication model is well suited to the dynamic establishment of communication relationships amongst application components during the lifetime of the application. This aided in the extensibility of the application, as sensors, actuators, and sentient objects could easily be added to, or removed from the application, easing application evolution.

The generation of code by the visual programming tool, combined with the load/save capability, meant that the application could be easily extended and the entire codebase rapidly regenerated, without the need to edit low-level code.

During the development of the sentient vehicle application these capabilities were used extensively in an incremental approach to development whereby additional functionality was added to the application, and periodic testing was carried out using a subset of sensor input.

Heterogeneity

The ability of heterogeneous devices to interact is crucial in pervasive computing applications, where large numbers of disparate hardware devices have to cooperate. The sentient vehicle application employed a set of heterogeneous hardware devices and system software which was largely transparent to the individual developers. For example, the vehicle sensors and actuator were developed in C++ and ran on Windows CE, whilst the traffic light sensor was developed in C++ running on Windows XP, and the vehicle sentient object was developed in Java running on Fedora. Developers of individual components were not required to be aware of what platform other components would execute on, and simply developed each component with regard to the relevant event types produced/consumed.

Reusability

Although the initial effort expended in the development of the sensor and actuator components involved interaction with device-level protocols, the resulting components are eminently re-usable. The components are decoupled and not tightly integrated into an individual application, and may easily be integrated into other applications. It is envisaged that in time, libraries of sensor and actuator components will emerge, reducing the need to create components from scratch for new applications.

Ad hoc interaction

The use of an event service specifically designed for ad hoc wireless networks provided a number of advantages. Chief amongst these was the ability to run the application anywhere without the need for a pre-installed designated service infrastructure. Consequent savings with regard to the cost and effort required to install and maintain network infrastructure were realised. The ability to run the application anywhere is essential in a truly pervasive computing environment.

Proximity based communication

The sentient vehicle application clearly demonstrates the benefits of proximity based event communication as offered by the STEAM event service. Communication and interaction between application components is often only of value when the components are located close together, and this fact may be used to minimise unnecessary communication. In the sentient vehicle application, the vehicle is only interested in receiving event notifications from the traffic light when it is within a certain proximity of the light, and furthermore, only when it is travelling *towards* the light. The event service employed in the sentient object model enabled such proximity-based filtering.

Sensor fusion

The probabilistic approach to sensor fusion adopted by the sentient object model provided a generic and systematic approach to managing the uncertainty of sensor data, which was easily adopted by the application developer. The visual programming tool was particularly effective in easing the specification of Bayesian networks.

A potential drawback of this approach was the need for extensive, time-consuming experimentation to determine prior probabilities for individual sensors on the vehicle, but this only had to be performed once, off-line.

6.2.7 Perspective

The sentient vehicle provided a valuable example of a mobile, context-aware application, the development of which was significantly eased by the sentient object model and associated programming tool. The application exhibited the important characteristics of loosely coupled application components, uncertain sensor data, and context-sensitive behaviour.

The application would have been more difficult to develop without the support offered by our programming model, and would have required the developer to write a substantial amount of low-level, procedural syntax. As it was, the programming tool removed the need for the sentient object developer to write any low-level procedural code for the capture, representation, and fusion of context data. The majority of the object behaviour was also

specified without the need to write any syntax, whilst complex-application specific behaviours were easily incorporated into the code generated by the programming tool, by editing a single file.

The resulting application was eminently and rapidly extensible due to the ability of the developer to load, edit, and re-generate a sentient object implementation using the programming tool. Individual components of the application may also be re-used in other applications due to the use of a loosely coupled generative communication mechanism, whilst new components may easily be added.

Extending the application

Although it was initially planned to have multiple sentient vehicles cooperating with each other in the application, construction of the hardware turned out to be a limiting factor.

6.3 Applicability of the approach

The application of the programming model to the representative scenarios described in this chapter allows for an evaluation of the overall approach provided by the sentient object model and associated programming tool, to the development of mobile, context-aware applications. Since the overall goal of the approach is to aid in the realisation of pervasive computing by easing application development, we evaluate the applicability of the approach through a discussion of the pre-requisite skills of developers, as well as the usefulness of the approach when considered in areas of pervasive computing other than mobile, context-aware applications. Finally the potential limitations of our approach are discussed with perspective on how these might be managed.

6.3.1 Requisite abilities

The sentient object model and associated programming tool provide a domain-specific approach to the development of mobile, context-aware applications. Whilst the high-level abstractions of the sentient object model, coupled with the capabilities of the programming tool

vastly reduce the complexity of application development, a certain level of technical ability is still required of users of the system. The target audience is thus experienced computer users who have had experience working with a range of applications, preferably high-level, domain-specific software engineering and CASE tools, and have an interest in the domain. Significantly, it is not necessary that users have programming experience with any specific low-level, procedural language, such as C++ or Java since there is no requirement to write any low-level code. The only part of the generated sentient object that may be edited by the developer - the rule-base, does not require procedural programming ability, nor is recompilation of the sentient object necessary following editing of the rules.

Although the approach provided still demands a moderate level of technical systems competence, abstraction away from the need to hand-code individual applications from scratch significantly improves the accessibility of mobile, context-aware application development to potential application developers. Experienced computer users typically have the ability to rapidly master new applications without the requirement for extensive training. More importantly such users have the confidence and often the desire to learn new systems, tools, and techniques. It is hoped that the approach will find particular value amongst the ubiquitous computing research community where much effort is currently spent on complex, low-level development which is often an orthogonal issue. Furthermore, users within this community have the significant advantage of understanding the domain for which the approach has been developed. Whilst the sentient object model and programming tool do not present any significant technical challenges, a thorough understanding of the domain-specific abstractions employed is necessary. A clear understanding of the concepts of context, sensors, actuators, and sensor fusion is essential and it is unlikely that the target audience will have difficulty with such concepts as they are core domain concepts. The approach of constructing Bayesian networks for fusing multi-modal sensor data, the use of production rules to govern behaviour, and the representation of context as a hierarchy, are more novel concepts that may not have been encountered even by experts within the domain. The provision of useable visual components for constructing Bayesian networks and production rules within the programming tool, as well the use of the context hierarchy to decompose application development, are designed

to facilitate the understanding of the overall approach to application development.

The *learning curve* is a term commonly used to express the relationship between competence in a task, against the time taken learning the task. A steep learning curve indicates a rapid increase in competence may be achieved for each unit of time spent learning the task, whilst a shallow learning curve indicates a more moderate increase in competence over time. The sentient object model and programming tool exhibit a steep learning curve when compared to other approaches to developing mobile, context-aware applications, since the high-level abstractions provided by the approach allow the rapid development of applications within a short period of time. For example, whilst it is necessary that users spend time learning and understanding the probabilistic approach to sensor fusion adopted by the model, they are not required to learn how to write low-level Bayesian network code, and are able to become productive more quickly.

6.3.2 Applicability to other types of application

By design, the sentient object model and programming tool are of most value applied to a specific class of application, namely mobile, context-aware applications. Nevertheless many of the abstractions adopted by the approach are common across the pervasive computing domain and may be applied to other types of pervasive computing applications. All facets of pervasive computing are predicated on interaction with the environment through sensing of data from the environment, and acting based on this data. Whilst our approach has focused specifically on mobile environments, the majority of domain abstractions we define are eminently applicable to stationary environments. A prime example is that of 'smart' environment applications, such as intelligent rooms and buildings where mobility is not a critical factor, but the core challenges of sensing, fusing sensor data, representing the data, making intelligent inferences, and taking action are of central importance. The sentient couch application described within this chapter clearly illustrates how our approach may be applied to such applications and it is our belief that the approach will find utility in other pervasive computing applications.

6.3.3 Limitations of the approach

Whilst our approach does significantly ease the development of mobile, context-aware applications, it does nonetheless have its own limitations. One potential limitation is the library of sensor and actuator components and associated descriptors, as described in sections 5.2 and 5.3 are not exhaustive, and may not contain all the components that a developer requires for a particular application. Although the library of sensor and actuator components may be growing rapidly, the risk remains that the development of a particular application is hampered by the lack of sensor or actuator components.

As discussed in section 4.2.1 the current version of STEAM only provides best-effort delivery semantics to application components that employ it as a communication mechanism. As a result, components should not expect reliable delivery of data and must be designed accordingly. Later releases of STEAM are expected to address this limitation.

Fostering widespread adoption of the sentient object model and programming tool by others outside the research community remains a significant challenge, and one that is characteristic of many tools developed in research laboratories. Whilst research into ubiquitous computing, and in particular context-aware computing has been ongoing for over a decade, much of the work remains in the research community with limited industrial activity. We believe the best way to address this potential limitation is through provision of free access to the programming tool, and championing of its use and evaluation through relevant platforms. To this end, the sentient object model and programming tool have been presented to the international research community through a variety of forums, resulting in significant interest from users already being generated.

Associated with the free distribution of the programming tool, the tool's future development is challenging, as it provides common functionality to a wide range of users with potentially disparate needs and expectations of the tool and generated code. The resolution of conflicting requests from the user community and the effective management of the evolution of the tool to ensure that it remains useful and relevant provide a significant challenge inherent in such an approach. Release of the programming tool as an open-source community project is one approach to managing this evolution.

6.4 Summary

This chapter evaluated our proposed programming model by applying it to the design and development of two exemplar context-aware applications. The sentient couch application, having been previously implemented following no structured methodology, offered the opportunity to compare this approach with that offered by the sentient object model. The sentient object model was found to offer the advantages of decoupled development allowing application components to be worked on by multiple developers simultaneously, as well as providing application extensibility whereby new components and functionality could easily be added to the application. The clear separation of inference rules which act on context information, from the rest of the application code, eased maintenance and evolution of application logic, without affecting the code controlling context capture. Furthermore, the application was not tightly linked to any particular hardware and devices could easily be substituted for others providing the same event interface.

The sentient vehicle application provided an example of an application which exploits the proximity based filtering capabilities of the event service incorporated in the sentient object model, as well as ad hoc communication between mobile application components. The model proved flexible enough for the design and implementation of the application, with particular advantages realised with regard to sensor fusion and specification of inference rules.

The next chapter presents our conclusions, based on the design and application of our programming model to a set of example applications.

Chapter 7

Conclusion

This thesis began by deriving a set of requirements for a programming model supporting the development of context-aware applications in mobile environments. A review of state of the art approaches to supporting the development of such applications revealed that no single approach currently provides comprehensive support for all the requirements derived. We described the sentient object model as a model that supports the development of mobile, context-aware applications, and the implementation of a graphical programming tool providing a high-level interface for programming sentient objects was described.

This chapter concludes the thesis by examining its major contributions, and discussing issues remaining open for further work.

7.1 Contribution

As concluded from our state-of-the-art review, there is currently no commonly-accepted programming model supporting the development of context-aware applications in mobile, ad hoc environments that provides generic support for the complete set of requirements identified and motivated during our overview of the research area.

The realisation of truly pervasive computing implies very widespread deployment of applications which require little or no human interaction. Key to this realisation are context-aware applications that are able to behave autonomously, in a proactive manner, based on context

information derived from a multitude of sensor inputs. The development of such applications remains complex and beyond the reach of average developers, hampering their widespread deployment, and consequently the realisation of the vision of pervasive computing as a whole.

As discussed in chapter 2, there are a number of challenges to developing context-aware applications in mobile, ad hoc environments, related to the capture, representation, and processing of context data. Existing approaches to application development discussed in chapter 3 recognise and typically address only a subset of these requirements and challenges. While comprehensive solutions exist to individual requirements such as the abstraction of sensor data, management of uncertain sensor data, and the processing of sensor data, until now there has been no *unified* approach to application development. Furthermore, most of the current approaches remain inaccessible to the majority of application developers, with minimal support offered to develop applications.

The main contribution of this thesis is a programming model for the development of context-aware applications in mobile, ad hoc environments that fulfills the major requirements identified of such a model. The programming model is based on the sentient object model that provides abstractions of sensor and actuator devices, and incorporates an event service specifically designed for mobile, ad hoc networks, to provide communication between loosely-coupled application components. Such anonymous, generative communication is inherently scalable, and provides for application extensibility. The sentient object abstraction provides a probabilistic approach to managing the uncertainty of sensor data, as well as a systematic and efficient approach to context representation and rule-based inference based on a hierarchy of contexts. Crucially, this support is offered to the application developer in an intuitive, accessible, and easy to use graphical programming tool that provides for high-level specification of a sentient object before generating low-level Java code implementing the object. In addition, the model defines language-independent descriptions of all the components, providing a template for alternative implementations. The automated generation of low-level code by the programming tool affords the advantages of high-quality, consistent, and easily customisable code, and allows more time to be spent on application design and testing. Furthermore, the generation of pure Java code by the programming tool ensures that

applications may run, unchanged, on a wide range of platforms.

The applications developed in chapter 6 serve to verify the applicability and usability of the sentient object model in the design and implementation of context-aware applications. The value of the model with respect to both application design and implementation was demonstrated through these applications. In the absence of the support offered by the sentient object model, the development of these applications would have been more difficult and would almost certainly have resulted in less extensible and maintainable applications. In the case of the sentient couch application, where a previous application had been implemented, the use of the sentient object model to design the application, and the programming tool to implement it resulted in a more concise implementation of the same functionality.

7.2 Future work

Based on the contribution of this thesis, there remain options for possible future work in the area. At present, the programming model exposes a single sensor fusion mechanism to application developers for managing the uncertainty of context data. Although this multi-modal fusion mechanism, based on probabilistic Bayesian networks, is suitably generic to be applied to a wide range of application scenarios, the provision of other approaches to managing the uncertainty of sensor data will only serve to reinforce the generality and utility of our programming model. Specifically, approaches to multi-modal sensor fusion based on Dempster-Schafer Theory as an extension to Bayesian probability, and multivariate Gaussian modelling, appear promising.

The language-independent template description of a sentient object in the form of an XML descriptor provides the opportunity to port sentient object implementations to other languages and platforms, by adding a module to the programming tool that generates the object specification in another language. Due to the use of an event service implemented on a range of diverse platforms, components running on heterogeneous platforms may easily inter-operate. A natural primary extension to the programming model could implement a sentient object specification in C++ for Windows CE, which is a widely adopted platform amongst mobile devices.

At present, the programming tool supports somewhat basic rule specification, based on fragments of context data in the sentient object. Complex custom behavioural rules still need to be hand written, and providing further support for the development of these complex behaviours, would be a useful extension to the tool. Furthermore, a number of exciting alternative approaches to inference exist, that deserve exploration within the programming model to assess their suitability for integration into the sentient object model. In particular, machine learning approaches, including reinforcement learning, and artificial neural networks provide promising alternative approaches to inference. Once again, the provision of a range of approaches to intelligent inference will only increase the value of the programming model.

Finally, the autonomy of sentient objects may be further reinforced through the adoption of algorithms to enable learning of behaviour, as well as algorithms providing autonomic, self-healing capabilities, further reducing the necessity for costly human control.

Appendix A

DTD for a sentient object XML descriptor

```
<?xml version='1.0' encoding='us-ascii'?>
<!ELEMENT actuator ( actuator-name, actuator-event+ ) >
<!ELEMENT actuator-event ( actuator-event-name, actuator-event-parameter+ ) >
<!ELEMENT actuator-event-name ( #PCDATA ) >
<!ELEMENT actuator-event-param-name ( #PCDATA ) >
<!ELEMENT actuator-event-param-type ( #PCDATA ) >
<!ELEMENT actuator-event-parameter ( actuator-event-param-name,
    actuator-event-param-type ) >
<!ELEMENT actuator-name ( #PCDATA ) >
<!ELEMENT actuators ( actuator+ ) >
<!ELEMENT behaviour-rules ( behaviour-rule+ ) >
<!ELEMENT behaviour-rule ( behaviour-rule-name ) >
<!ELEMENT behaviour-rule-name ( #PCDATA ) >
<!ELEMENT context ( context-event*, context-name, context-type, rules?,
    network?, transition-context* )* >
<!ELEMENT context-event ( context-event-name, context-event-param-filter+ ) >
<!ELEMENT context-event-name ( #PCDATA ) >
<!ELEMENT context-event-param-filter ( param-name, filter-operator,
    filter-operand ) >
<!ELEMENT context-name ( #PCDATA ) >
<!ELEMENT context-type ( #PCDATA ) >
<!ELEMENT contexts ( context+ ) >
<!ELEMENT definition ( for | given | table )* >
<!ELEMENT filter-operand ( #PCDATA ) >
<!ELEMENT filter-operator ( #PCDATA ) >
<!ELEMENT for ( #PCDATA ) >
<!ELEMENT given ( #PCDATA ) >
...
```

```
...
<!ELEMENT input-object ( input-object-event+ ) >
<!ELEMENT input-object-event ( input-object-event-name,
    input-object-event-parameter+ ) >
<!ELEMENT input-object-event-name ( #PCDATA ) >
<!ELEMENT input-object-event-param-name ( #PCDATA ) >
<!ELEMENT input-object-event-param-type ( #PCDATA ) >
<!ELEMENT input-object-event-parameter ( input-object-event-param-name,
    input-object-event-param-type ) >
<!ELEMENT input-objects ( input-object+ ) >
<!ELEMENT inputs ( sensors?, input-objects? ) >
<!ELEMENT name ( #PCDATA ) >
<!ELEMENT network ( name, (variable | definition)* ) >
<!ELEMENT object ( object-name, inputs, contexts, outputs ) >
<!ELEMENT object-name ( #PCDATA ) >
<!ELEMENT outcome ( #PCDATA ) >
<!ELEMENT output-object ( output-object-event+ ) >
<!ELEMENT output-object-event ( output-object-event-name,
    output-object-event-parameter+ ) >
<!ELEMENT output-object-event-name ( #PCDATA ) >
<!ELEMENT output-object-event-param-name ( #PCDATA ) >
<!ELEMENT output-object-event-param-type ( #PCDATA ) >
<!ELEMENT output-object-event-parameter ( output-object-event-param-name,
    output-object-event-param-type ) >
<!ELEMENT output-objects ( output-object+ ) >
<!ELEMENT outputs ( actuators?, output-objects? ) >
<!ELEMENT param-name ( #PCDATA ) >
<!ELEMENT rules ( transition-rules?, behaviour-rules? ) >
<!ELEMENT sensor ( sensor-name, sensor-event+ ) >
<!ELEMENT sensor-event ( sensor-event-name, sensor-event-parameter+ ) >
<!ELEMENT sensor-event-name ( #PCDATA ) >
<!ELEMENT sensor-event-param-name ( #PCDATA ) >
<!ELEMENT sensor-event-param-type ( #PCDATA ) >
<!ELEMENT sensor-event-parameter ( sensor-event-param-name,
    sensor-event-param-type ) >
<!ELEMENT sensor-name ( #PCDATA ) >
<!ELEMENT sensors ( sensor+ ) >
<!ELEMENT table ( #PCDATA ) >
<!ELEMENT transition-context ( #PCDATA ) >
<!ELEMENT transition-rules ( transition-rule+ ) >
<!ELEMENT transition-rule ( transition-rule-name ) >
<!ELEMENT transition-rule-name ( #PCDATA ) >
<!ELEMENT variable ( name, outcome+ ) >
<!ATTLIST variable type NMTOKEN #REQUIRED>
```

Listing A.1: DTD for an XML sentient object descriptor

Appendix B

Example components

B.1 Actuator

An actuator which sends an e-mail message in response to the consumption of an event is illustrated in Listings B.1 and B.2, and described below.

B.1.1 SMTP actuator

The SMTP actuator class listed in Listing B.1 consumes one type of event, shown in Table B.1. The event contains two parameters, namely the address to which to send the mail notification, and a short message to include in the notification e-mail. The actuator class creates an instance of `SMTPActuatorDeliveryCallback` class, where most of the processing is performed.

B.1.2 SMTP actuator delivery callback

The delivery callback class illustrated in Listing B.2 extends the `ActuatorDeliveryCallback` class provided by the programming tool, and has one method, `deliver()` to be extended by

Event type	Parameter 0	Parameter 1
SMTP	`S_STR` address	`S_STR` message

Table B.1: Event type consumed by `SMTPActuator`

```
import ie.tcd.cs.dsg.jsteam.*;

public class SMTPActuator extends Actuator{
  public static void main(String[] args){
    SMTPActuator actuator = new SMTPActuator();
    /** Actuator is in a fixed position(5320.00, -615.008),
      * with range set to 70, periodicity to 1 */
    actuator.startActuator(70, 1, 5320.00, -615.008,
                           new SMTPActuatorDeliveryCallback());
    actuator.subscribe("SMTP");
  }
}
```

Listing B.1: `SMTPActuator.java`

Event type	Parameter 0
Barcode	`S_STR` code

Table B.2: Event type produced by `BarcodeSensor`

the developer, and which specifies the behaviour that occurs when an event is delivered. In the case of this actuator, the class extracts the event parameters, opens an SMTP connection, and sends an email to the specified recipient.

B.2 Sensor

A sensor that reads the output of a barcode reader, and publishes one type of event containing the number represented by the barcode. The sensor may be easily extended to provide output at a higher level of abstraction, e.g, to perform a database lookup and return a name associated with the barcode identifier.

B.2.1 Barcode sensor

The barcode sensor consists of one class, illustrated in Listing B.3, and produces a single type of event, as shown in Table B.2.

```
import ie.tcd.cs.dsg.jsteam.*;
import sun.net.smtp.SmtpClient;
import java.io.*;

public class SMTPActuatorDeliveryCallback extends ActuatorDeliveryCallback{
  /**
   * Main event handler method, called when the actuator receives
   * an event notification
   *
   *@param eventInstance the event notification
   */
  public void deliver(SC_dsEvent eventInstance){
    String mailfrom = "couch@cs.tcd.ie";
    String mailto = eventInstance.parValSTR(0);
    String content = eventInstance.parValSTR(1);
    try {
      SmtpClient smtp = new SmtpClient("mail.cs.tcd.ie");
      // Sets the originating e-mail address
      smtp.from("couch@cs.tcd.ie");
      // Sets the recipients' e-mail address
      smtp.to(mailto);
      // Create an output stream to the connection
      PrintStream msg = smtp.startMessage();
      msg.println("To: " + mailto);
      msg.println("From: " + mailfrom);
      msg.println("Subject: Sentient couch notification");
      msg.println(content);
      // Close the connection to the SMTP server and send the message
      smtp.closeServer();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }
  }
}
```

Listing B.2: `SMTPActuatorDeliveryCallback.java`

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.InputStreamReader;
import java.io.BufferedReader;
import ie.tcd.cs.dsg.jsteam.*;

public class BarcodeSensor extends Sensor implements Runnable{
  private SP_dsEventType barcodeEventType; //The sensor event type

  /** Constructor */
  public BarcodeSensor(){
    //Create a circular proximity of size 20m
    super(new SP_Circle(20));
  }

  public void run(){
    //Wait for input from the barcode reader
    while(true){
      try{
        InputStreamReader reader = new InputStreamReader(System.in);
        BufferedReader stdin = new BufferedReader(reader);
        //Publish the event when input is received
        publishEvent(stdin.readLine());
      }catch(Exception e){
        e.printStackTrace();
      }
      try{
        Thread.sleep(1000);
      }catch (InterruptedException e){}
    }
  }

  /**
   * Creates the event instance containing one parameter - the barcode
   * and calls the super class to publish the event
   *
   *@param  code the barcode read by the hardware device
   */
  public void publishEvent(String code){
    S_ParameterValue[] pv = new S_ParameterValue[1];
    pv[0] = new SP_ParameterValueSTR(code);
    SP_dsEvent eventInstance = new SP_dsEvent(barcodeEventType.subject(), 1, pv,
        barcodeEventType);
    super.publishEvent(eventInstance);
  }

   /** Main method */
  public static void main(String args[]){
    BarcodeSensor sensor = new BarcodeSensor();
    S_EventParameterDeclaration[] epd = new S_EventParameterDeclaration[1];
    epd[0] = new S_EventParameterDeclaration("Code", JSteamConstants.S_STR);
    sensor.barcodeEventType = new SP_dsEventType("Barcode", 1, epd);
    sensor.startSensor(70, 1, 5320.00, -615.008);
    sensor.announceEventType(sensor.barcodeEventType);
    new Thread(sensor).start();
  }
}
```

Listing B.3: `BarcodeSensor.java`

Appendix C

Sentient couch application

C.1 Hardware

The sensors fitted to the couch are LPX 100 industrial transducers supplied by Precision Transducers Ltd.[1] and illustrated in Figure C.1. Important characteristics of this type of sensor are listed in Table C.1, and it is evident from these characteristics that the LPX 100 provides an accurate and inexpensive solution for the sensing of mass in the range of that of an average human.

The legs of the couch are fitted with custom-made steel shoes manufactured by a local engineering firm and encasing the load sensors. This prevents movement of the couch leg over the sensor, as well as preventing slippage of the sensor on the laboratory floor.

[1] http://www.precisiontransducers.com

Fig. C.1: LPX 100 industrial load sensor

Capacity	100 kg
Nominal output at capacity	2.0mV/V +/ − 1%
Safe load	150 kg
Linearity error†	0.1%
Price	USD 103.50

Table C.1: Characteristics of the LPX 100 load sensor
†Deviation of sensor output from expected output at a known load

Fig. C.2: The sentient couch in use

Each of the LPX 100 transducers is connected to a PT650D weighing indicator, sourced from Chi Mei Electronics in Hong Kong. The weighing indicator converts the analogue signal from the LPX 100 to a digital signal, amplifies it and outputs the signal to a PC on an RS232 interface. The PC has a Pentium processor, running Debian GNU/Linux.

At present, there is no specialised actuator hardware fitted to the couch, with actuation performed through a combination of software commands, and standard speakers and display monitors. The sensor and actuator components were developed by Mélanie Bouroche of the Distributed Systems Group.

Appendix D

Sentient vehicle application

D.1 Hardware

The sentient vehicle is a consumer-grade model radio control (RC) 1:6 scale Ford F150 Thunder purchased from a local electronics store[1], which has been augmented with a number of sensors, as well as a handheld computer and microcontroller. Our aim was to use low-cost, readily available components as far as possible. The use of cheap and readily available components in context-aware systems contributes to widespread adoption, aiding the realisation of truly pervasive computing. The hardware and sensor and actuator component software were developed according to the sentient object model, by Neil O'Connor of the Distributed Systems Group [O'C04].

SRF08 Ultrasonic range finder

Ultrasonic sensors are commonly used for a variety of distance or proximity measurements. An ultrasonic sensor works by transmitting short bursts of high frequency sound towards a target, and measuring the time taken for the reflected echo to return to the sensor. The distance to the target is then measured using the time of the echo and the speed of sound. Since ultrasonic sensors depend on the speed of sound for accurate range determination, their accuracy is influenced to varying degrees by environmental factors, such as temperature, air

[1] http://www.maplin.co.uk

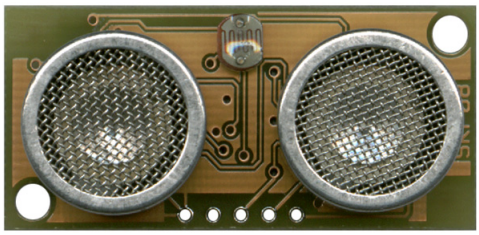

Fig. D.1: Devantech SRF08 ultrasonic range-finder

Voltage	$5v$
Current	15mA typ. 3mA standby
Frequency	40KHz
Range	3cm - 600cm
Connection	I2C Bus
Timing	Fully timed echo
Echo	Multiple echo
Unit of measure	μS, mm or inches
Dimensions	43mm w x 20mm d x 17mm h
Accuracy	3-4 cm
Cost	USD 45

Table D.1: Devantech SRF08 characteristics

pressure and turbulence, humidity and acoustic interference [Shi89], introducing elements of uncertainty into the readings taken by this type of sensor.

We incorporate Devantech [2] SRF08 type ultrasonic range finders in our sentient vehicle to provide range determination and obstacle avoidance capabilities to the vehicle. The SRF08 is shown in Figure D.1, whilst the characteristics of this type of sensor are illustrated in Table D.1. It can be seen that this type of sensor claims to measure ranges of up to six meters with a high degree of accuracy and at a low cost.

The SRF08 range finder stores a set of up to seventeen ranges (that is the distance to a detected obstacle) from each single ranging operation in a set of 36 registers. When a ranging

[2] http://www.robot-electronics.co.uk

Register no.	Read	Write
0	Software revision	Command register
1	Light sensor	Max gain register (default 31)
2	1^{st} echo high byte	Range register (default 255)
3	1^{st} echo low byte	N/A
34	17^{th} echo high byte	N/A
35	17^{th} echo low byte	N/A

Table D.2: Devantech SRF08 registers

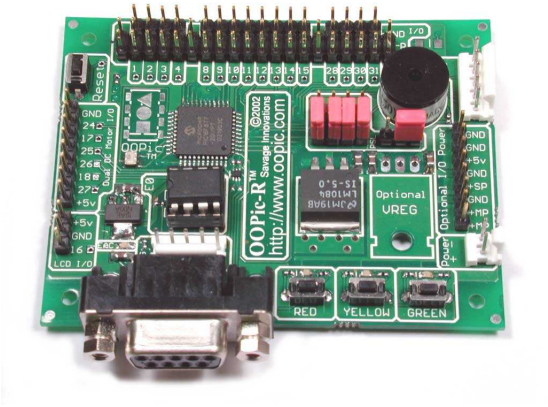

Fig. D.2: OOPic-R microcontroller board

command is received over the I2C bus by the sensor, an ultrasonic transmission is made. The sensor then listens for echoes up to the maximum range of its transmission ($65ms$). The sensor does not respond to any I2C communication during this time, freeing a controller from the need to use a timer. Once ranging is complete, measured ranges may be read by referring to the appropriate registry entry as illustrated in Table D.2. Locations 2 - 35 contain the readings for the last ranging command, with each range represented by a high and a low byte. Location 1 stores the value of a light sensor integrated into the SRF08.

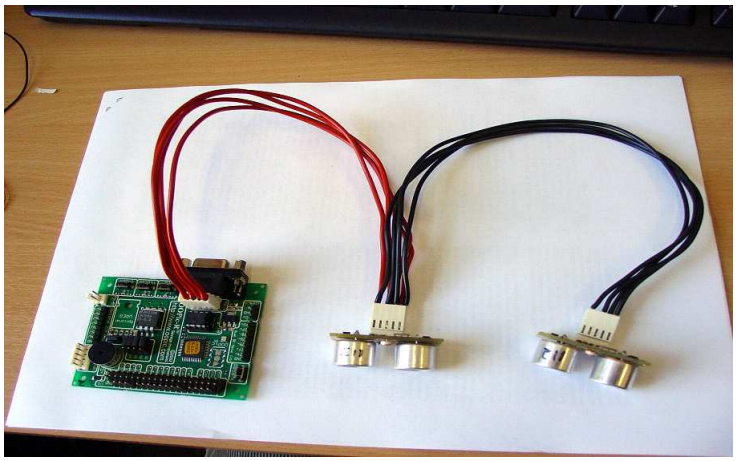

Fig. D.3: 2 SRF08 sensors connected to an OOPic-R microcontroller via an I2C bus [O'C04]

OOPIC-R microcontroller

In order to interface the SRF08 sensors with an iPAQ, a programmable microcontroller was required, which could communicate via I2C with the SRF08 sensors and preferably via RS-232 to the handheld computer (since the handheld has two RS-232 ports available). A commercially available microcontroller meeting these requirements is the Object-Oriented Programmable Interrupt Controller (OOPic) 'R' board manufactured by Savage Innovations[3]. In addition to providing the I2C to RS-232 capability, the OOPIC-R board, illustrated in Figure D.2 also provides sixteen digital I/O lines with power and ground connections which can be used to interface hardware actuators to an RS-232 connection.

The OOPic-R is programmed on a PC in an object-oriented manner using a freely downloadable development environment that supports Basic, Java or C-like syntax. Compiled OOPic binary is uploaded to the board via an RS-232 connection.

HP iPAQ 5550

An HP iPAQ 5550 is used to control the sensors and the actuators. The iPAQ is a handheld PC with a 400MHz processor and 128MB of RAM, as well as a built-in 802.11b wireless

[3]http://www.oopic.com

Fig. D.4: Magellan GPS receiver

interface and a PCMCIA card exporting two RS-232 interfaces.

Magellan GPS receiver

The vehicle is equipped with a Magellan GPS receiver, illustrated in Figure D.4 which enables the vehicle to be aware of its location and additionally provides location information to the STEAM location service. The GPS receiver provides location updates in the form of National Marine Electronics Association (NMEA) 0183 Interface Standard. to the handheld computer via an RS-232 interface.

Electronic compass

The vehicle is equipped with a low-cost, I2C bus CMPS03 electronic magnetic compass sourced from Devantech[4], which provides information about what direction the vehicle is heading in, and is interfaced with the OOPIC-R microcontroller via an I2C bus.

D.2 Navigational formulae

The navigational formulae used within the sentient car are based on great circle navigation [Ste00]. The GPS location data consumed by the car in the form of events from the location service, consists of longitude/latitude pairs in NMEA-0183 format (as output by the GPS

[4]http://www.robot-electronics.co.uk

receiver), in units **ddmm.mmmm**, where dd represents degrees, mm minutes, and .mmmm decimal minutes. Before these coordinates may be used in the navigational formulae, they must be converted to degrees and decimal degrees, and then to radians. The process of converting the coordinates to radians is as follows

1. Divide **ddmm.mmmm** by 100 to yield **dd**
2. Divide **mm.mmmm** by 60 to yield **.dddd**
3. Add **dd** to **.dddd** to yield **dd.dddd**
4. Divide **dd.dddd** by 57.2957795 to yield radians

D.2.1 Distance between two points

The distance between two points represented by the pairs $< lat1, lon1 >$ and $< lat2, lon2 >$ is calculated according to the following equation

$$distance = \arccos(\sin(lat1) \times \sin(lat2) + \cos(lat1) \times \cos(lat2) \times \cos(lon1 - lon2)) \quad \text{(D.1)}$$

D.2.2 Bearing between two points

To calculate the bearing[5] between two points represented by the pairs $< lat1, lon1 >$ and $< lat2, lon2 >$, the great circle distance between the points must first be calculated according to equation (D.1) The equation to calculate the bearing between two points is

$$bearing = \arccos\left(\frac{\sin(lat2) - \sin(lat1) \times \cos(distance)}{\cos(lat1) \times \sin(distance)}\right) \quad \text{(D.2)}$$

This equation fails with a point located at the poles, but is sufficient for non-polar navigation, and thus adequate for the needs of the sentient car.

[5]Defined as the angle measured horizontally from north to current direction of travel

Fig. D.5: Sentient vehicle hardware

D.3 Experimental setup

The combined hardware setup of the sentient vehicle is illustrated in Figure D.5. The location of the three forward facing, and two side facing ultrasonic sensors may clearly be seen. In addition, there is a single rear-facing ultrasonic sensor mounted at the back of the vehicle, partially obscured by the iPAQ. The iPAQ and GPS receiver are mounted towards the rear of the vehicle, whilst the OOPIC and radio control boards are mounted out of sight inside the vehicle, together with battery packs powering the boards.

D.3.1 Forward obstacle avoidance

The prototypical sentient vehicle application calls for the avoidance of obstacles located in the path of the vehicle whilst it is travelling forward. Figure D.6 illustrates the coverage of the forward facing sensors, and the area that must be free of obstacles for the vehicle to proceed forward.

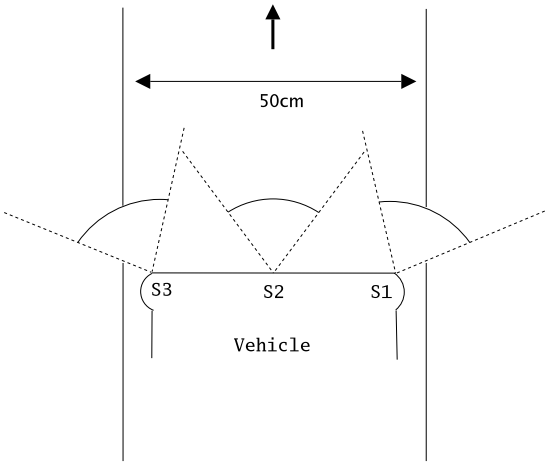

Fig. D.6: Configuration of forward facing ultrasonic sensors

Sensor fusion

Whilst any of the 3 forward facing sensors is able to detect the presence of an obstacle within its 'cone' of vision, by fusing the outputs of the 3 individual sensors, the position of the obstacle may be more accurately postulated, and more intelligent actions taken.

The sentient object model adopts an approach to sensor fusion based on probabilistic Bayesian networks which require the specification of prior probabilities to construct them. In the case of the ultrasonic sensors mounted on the sentient vehicle, the probability that the sensors *detected* an obstacle at a particular distance, given that the obstacle is located that distance away from the sensor.

Experiments were conducted using the experimental setup illustrated in Figure D.7. The various locations of obstacle for which probabilities were calculated in this experiment are illustrated in Figure D.8, and are as follows:

1. Scenario A - an obstacle is located directly in front of the vehicle. In this scenario, the vehicle could stop, turn left, or turn right

2. Scenario B - an obstacle is located to the left of the vehicle, but still in the path of the vehicle. In this case, the vehicle should turn to the right to avoid the obstacle

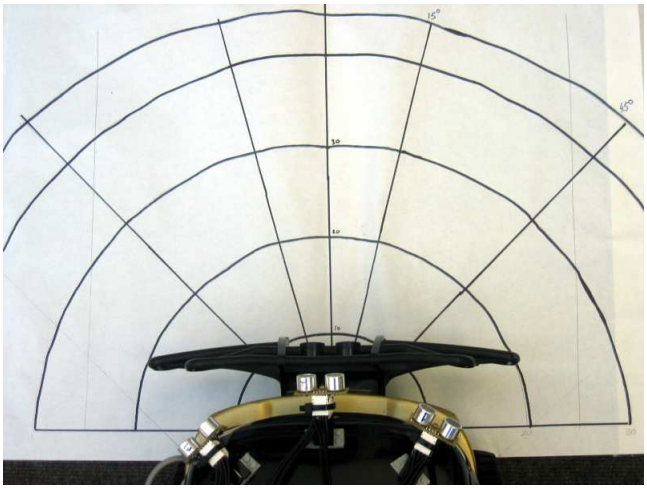

Fig. D.7: Experimental setup to determine prior probabilities for forward facing sonar sensors

3. Scenario C - an obstacle is located to the right of the vehicle, but still in its path. In this case, the vehicle should turn to the left to avoid the obstacle

4. Scenario D - An obstacle is located to the left of the vehicle, but outside it's path, and thus no action is necessary

5. Scenario E - An obstacle is located to the right of the vehicle, but outside it's path, and thus no action is necessary

In the first prototype of the application, the assumption was made that obstacles have a flat face, and are positioned perpendicular to the face of the middle sensor. This assumption was made due to the fact that when an obstacle is angled away from the face of the sensor, ultrasonic signals are not reflected back to the sensor, and the obstacle becomes 'invisible'.

Through iteration over the various obstacle placements, and recording of sensor readings, the set of prior probabilities for the three forward-facing ultrasonic sensors was calculated, as illustrated in Table 6.14. These values capture the conditional probabilities that each sensor detects an obstacle, given individual obstacle placements.

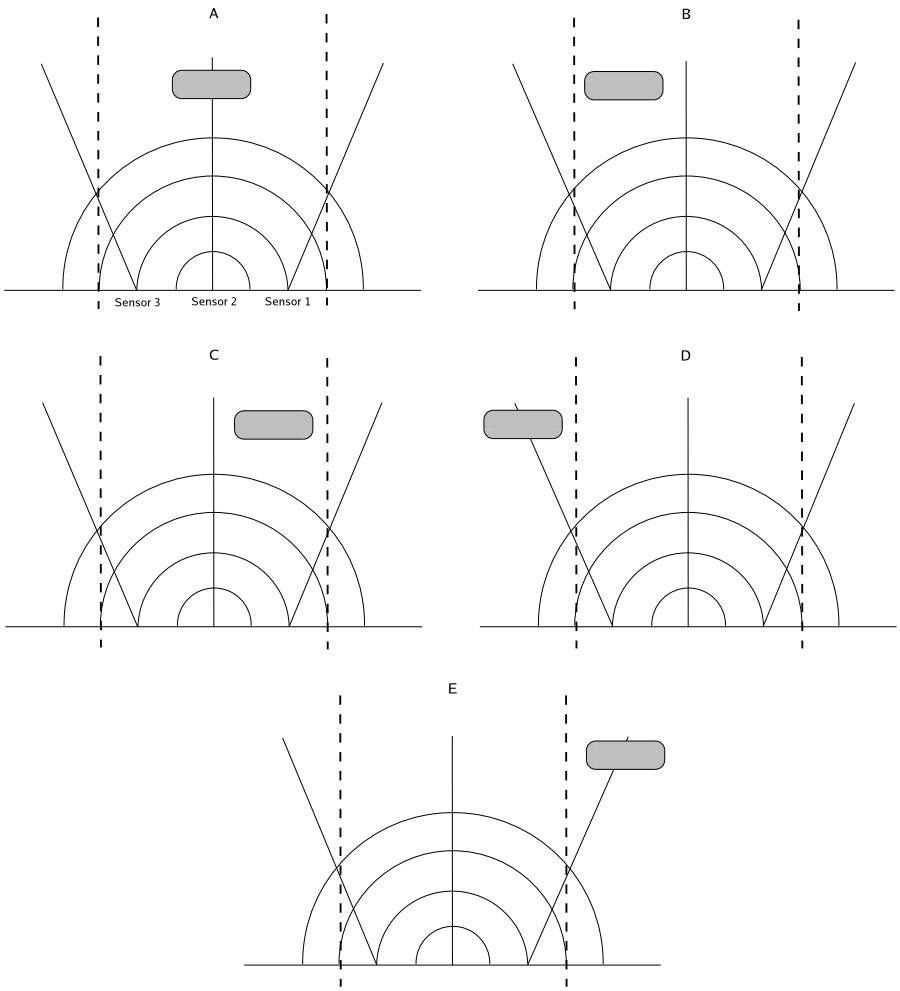

Fig. D.8: Obstacle positions tested in experiments

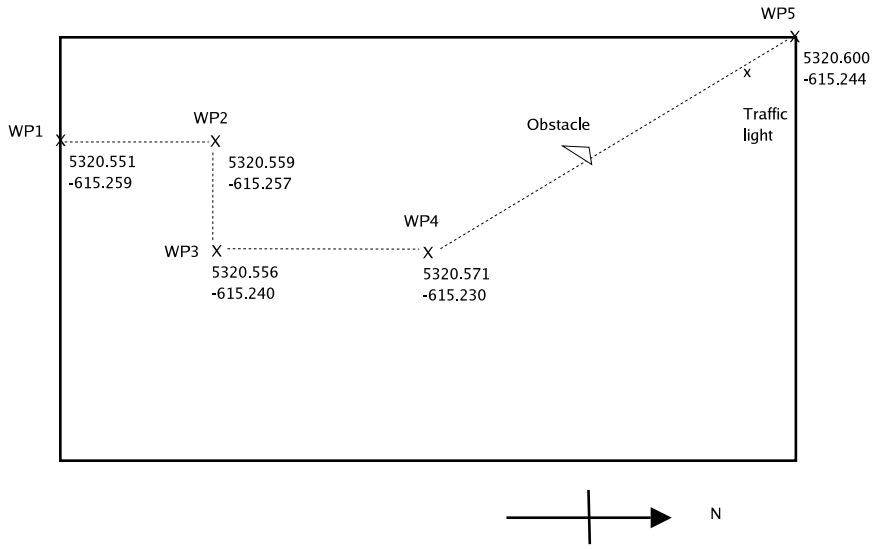

Fig. D.9: Waypoint navigation course

D.3.2 Traffic signal obeyance

The sentient vehicle subscribes to events produced by a traffic light sensor, which periodically changes status between red and green. The sensor publishes the events within a proximity of 20 metres, and the vehicle filters event notifications, so that notifications are only delivered when the vehicle is *approaching* the traffic signal. The code snippet below illustrates how an event filter is specified to provide this functionality within the application.

```
TrafficSignalFilter tsFilter = new SC_ConjunctiveContentFilter();
tsFilter.addTermPOS(0, JsteamConstants.SC_DISTANCE_DECREASES);
```

D.3.3 Waypoint navigation

In order to test autonomous navigation, a course was designed with five waypoints defined, as illustrated in Figure D.9. The goal was to follow the course defined by the edges linking these waypoints.

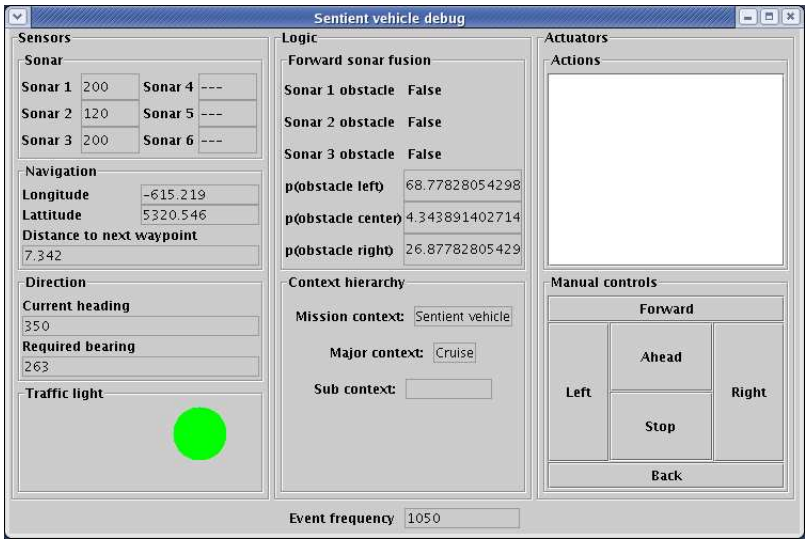

Fig. D.10: Car debug interface

D.3.4 Testing and debugging the application

In order to assist in debugging the application, a visualisation interface was constructed, as illustrated in Figure D.10 that subscribed to all events produced by the sensors on the car, as well as events produced by the sentient object. This gave the application developer the ability to see all event communication in the system at a glance. In addition to live outdoor testing of the system, a simulator was developed that produced simulated event streams from the vehicle sensors, in order to test and debug the vehicle sentient object. This simulator parsed an input file to provide periodic event notifications, describing a vehicle journey.

Bibliography

[AAJ+97] G.D. Abowd, C.G. Atkeson, Hong J., S. Long, R. Kooper, and M. Pinkerton. Cyberguide: A Mobile Context-Aware Tour Guide. *ACM Wireless Networks*, 3:421–433, 1997.

[ACH+01] Mike Addlesee, Rupert Curwen, Steve Hodges, Joe Newman, Pete Steggles, Andy Ward, and Andy Hopper. Implementing a Sentient Computing System. *IEEE Computer*, 34(8):50–56, August 2001.

[ACVS02] G. Ahn, A. Campbell, A. Veres, and L. Sun. SWAN: Service differentiation in stateless wireless ad hoc networks. In *Proceedings of IEEE INFOCOM*, New York, NY, USA, June 2002.

[Alb02] Albrecht Schmidt. *Ubiquitous Computing - Computing in Context*. PhD thesis, Lancaster University, November 2002.

[BAI93] B. R. Badrinath, Arup Acharya, and Tomasz Imielinski. Impact of mobility on distributed computations. *Operating Systems Review*, 27(2):15–20, 1993.

[Bay58] Thomas Bayes. An essay towards solving a problem in the doctrine of chances (reprint of 1763). *Biometrika*, 45:293–315, 1958.

[BBC97] 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, October 1997.

[BHK04] Mark H. Butler, Johan Hjelm, and Kazuhiro Kitagawa. CC/PP Information Page. `http://www.w3.org/Mobile/CCPP/`, 2004.

[BLC95] T. Berners-Lee and D. Connolly. RFC 1866 - Hypertext Markup Language version 2.0, November 1995.

[BLFF96] T. Berners-Lee, R. Fielding, and H. Frystyk. RFC 1945 - Hypertext Transfer Protocol – HTTP/1.0, May 1996.

[BMB+00] J. Bacon, K. Moody, J. Bates, R. Hayton, C. Ma, A. McNeil, O. Seidel, and M. Spiteri. Generic Support for Distributed Applications. *IEEE Computer*, 33(3):68–76, 2000.

[Bor00] G. Borriello. The challenges to invisible computing. *IEEE Computer*, 33(11):123–125, November 2000.

[Bro96] P. Brown. The stick-e document: a framework for creating context-aware applications. *Electronic Publishing*, 9(1):1–14, September 1996.

[BSAK95] Hari Balakrishnan, Srinivasan Seshan, Elan Amir, and Randy H. Katz. Improving TCP/IP Performance over Wireless Networks. In *Proceedings of 1st ACM Intl Conference on Mobile Computing and Networking (Mobicom)*, November 1995.

[BW95] B. R. Badrinath and Girish Welling. Event delivery abstraction for mobile computing. Technical Report LCSR-TR-242, 1995.

[Cal03] Rob Callan. *Artificial Intelligence*. Palgrave Macmillan, 2003.

[Can02] John Canny. Some Techniques for Privacy in Ubicomp and Context-aware Applications. In *Proceedings of Workshop on Socially-informed Design of Privacy-enhancing Solutions in Ubiquitous Computing, UbiComp 2002*, G'oteborg, Sweden, September 2002.

[CCKM01] Paul Castro, Patrick Chiu, Ted Kremenek, and Richard Muntz. A Probabilistic Location Service for Wireless Network Environments. In *Proceedings of Ubicomp 2001: Ubiquitous Computing: 3rd International Conference, Atlanta, GA, USA*, pages 18–34, September 2001.

[CDM+00] Keith Cheverst, Nigel Davies, Keith Mitchell, Adrian Friday, and Christos Efstratiou. Developing a Context-aware Electronic Tourist Guide: Some Issues and Experiences. In *Proceedings of CHI 2000 Conference on Human Factors in Computing Systems*, pages 17–24, The Hague, The Netherlands, April 1-6 2000.

[CDME00] Keith Cheverst, Nigel Davies, Keith Mitchell, and Christos Efstratiou. Using Context as a Crystal Ball: Rewards and Pitfalls. In *Proceedings of Workshop on Situated Interaction in Ubiquitous Computing, CHI 2000*, pages 17–24, The Hague, The Netherlands, April 1-6 2000.

[CDMF99] Keith Cheverst, Nigel Davies, Keith Mitchell, and Adrian Friday. The Role of Connectivity in Supporting Context-Sensitive Applications. In H.W. Gellersen, editor, *Handheld and Ubiquitous Computing, Lecture Notes in Computer Science*, 1707, pages 193–207, Heidelberg, 1999. Springer-Verlag.

[CGS+03] 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. *Pervasive Computing Magazine*, 2(3):52–61, 2003.

[CHRC01] Renato Cerqueira, Christopher K. Hess, Manuel Román, and Roy H. Campbell. Gaia: A Development Infrastructure for Active Spaces. In *Proceedings of Workshop on Application Models and Programming Tools for Ubiquitous Computing (held in conjunction with the UBICOMP 2001)*, Atlanta, Georgia, USA, September 2001.

[CK00] Guanling Chen and David Kotz. A Survey of Context-Aware Mobile Computing

Research. Technical Report TR2000-381, Dept. of Computer Science, Dartmouth College, November 2000.

[CK02a] Guanling Chen and David Kotz. Context aggregation and dissemination in ubiquitous computing systems. In *Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, pages 105–114. IEEE Computer Society Press, June 2002.

[CK02b] Guanling Chen and David Kotz. Solar: A pervasive computing infrastructure for context-aware mobile applications. Technical Report TR2002-421, Department of Computer Science, Dartmouth College, February 28 2002.

[CK02c] Guanling Chen and David Kotz. Solar: An open platform for context-aware mobile applications. In *Proceedings of the First International Conference on Pervasive Computing (Short paper)*, pages 41–47, Zurich, Switzerland, June 2002.

[CK03] Guanling Chen and David Kotz. Context-sensitive resource discovery. In *Proceedings of the First IEEE International Conference onPervasive Computing and Communications (Percom 2003)*, pages 243–252, Forth Worth, Texas, USA, March 2003.

[C.L82] Forgy C.L. Rete: A fast Algorithm for the Many Pattern / Many Object Pattern Match Problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[CLK04] Guanling Chen, Ming Li, and David Kotz. Design and Implementation of a Large-Scale Context Fusion Network. In *Proceedings of the First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous 2004)*, Boston, MA, USA, August 2004.

[CM99] P. Castro and R. Muntz. Using Context to Assist in Multimedia Object Retrieval Applications. In *Proceedings of ACM Workshop on Multimedia Intelligent Storage and Retrieval Management*, Orlando, FL., USA, October 1999.

[CM00] Paul Castro and Richard Muntz. Managing Context Data for Smart Spaces. *IEEE Personal Communications*, 7:44–46, October 2000.

[CMD99] Keith Cheverst, Keith Mitchell, and Nigel Davies. Design of an Object Model for a Context Sensitive Tourist GUIDE. *Computers and Graphics*, 23(6):883–891, 1999.

[CMMM00] Paul Castro, Murali Mani, Siddhartha Mathur, and Richard Muntz. Managing Context for Internet Videoconferences: The Multimedia Interent Recorder and Archive. In *Proceedings of IS&T SPIE Conf. on Multimedia Computing and Networking 2000 (MMCN 2000)*, San Jose, CA, USA, January 2000.

[CNF01] G. Cugola, E. D. Nitto, and A. Fuggetta. The JEDI Event-Based Infrastructure and its Application to the Development of the OPSS WFMS. *IEEE Transactions on Software Engineering (TSE)*, 27:827 – 850, 2001.

[Cor03] Intel Corporation. Intel Centrino^{TM} Mobile Technology Performance Brief. `http://www.intel.com/performance/`, September 2003.

[Coy92] Peter Coy. Big Brother, Pinned to your chest. *Business Week*, (3279), August 1992.

[CP94] Perkins C. and Bhagwat P. Highly Dynamic Destination-Sequence DistanceVector Routing (DSDV) for Mobile Computers. *ACM SIGCOMM Computer Communication Review*, 24(4):234–244, October 1994.

[CRW01] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19:283 – 331, 2001.

[CSG99] D. Chen, A. Schmidt, and H.-W. Gellesen. An Architecture for Multi-Sensor Fusion in Mobile Environments. In *Proceedings of International Conference on Information Fusion*, volume II, pages 861–868, Sunnyvale, CA, USA, July 1999.

[CWKS97] B.P. Crow, I. Widjaja, J.G. Kim, and P.T. Sakai. IEEE 802.11 Wireless Local Area Networks. *IEEE Communications Magazine*, pages 116–126, September 1997.

[DA99] Anind K. Dey and Gregory D. Abowd. Towards a Better Understanding of Context and Context-Awareness. Technical Report GIT-GVU-99-22, Georgia Institute of Technology, June 1999.

[DFSA99] Anind K. Dey, Masayasu Futakawa, Daniel Salber, and Gregory D. Abowd. The Conference Assistant: Combining Context-Awareness with Wearable Computing. In *Proceedings of the 3rd International Symposium on Wearable Computers (ISWC '99)*, pages 21–28, San Francisco, CA, USA, October 1999.

[dIn99] Diego López de Ipiña. TRIP: A Distributed vision-based Sensor System *Ph.D. 1st Year Report*, LCE, Cambridge University Engineering Department, UK, 31 August 1999.

[dIn00] Diego López de Ipiña. Building Components for a Distributed Sentient Framework with Python and CORBA. In *Proceedings of the 8th International Python Conference*, Arlington, VA, USA, January 24-27 2000.

[dIn01] Diego López de Ipiña. Video-Based Sensing for Wide Deployment of Sentient Spaces. In *Proceedings of 2nd PACT Workshop on Ubiquitous Computing and Communications*, Barcelona, Spain, September 9 2001.

[dIn02] Diego López de Ipiña. *Visual Sensing and Middleware Support for Sentient Computing*. PhD thesis, Downing College, University of Cambridge, Cambridge, United Kingdom, January 2002.

[dInK01] Diego López de Ipiña and Eleftheria Katsiri. An ECA Rule-Matching Service for Simpler Development of Reactive Applications. *IEEE Distributed Systems Online*, 2(7), November 2001.

[dInL01] Diego López de Ipiña and Sai-Lai Lo. Sentient Computing for Everyone. In *Proceedings of the 3rd International Conference on Distributed Applications and Interoperable Systems (DAIS'2001)*, Krakow,Poland, September 17-19 2001.

[dInMH02] Diego López de Ipiña, Paulo Medonca, and Andy Hopper. TRIP: a Low-Cost Vision Based Location System for Ubiquitous Computing. *Personal and Ubiquitous Computing*, 6(3):206–219, May 2002.

[DMAC02] Anind K. Dey, Jennifer Mankoff, Gregory D. Abowd, and Scott Carter. Distributed Mediation of Ambiguous Context in Aware Environments. In *Proceedings of the 15th Annual Symposium on User Interface Software and Technology (UIST 2002)*, pages 121–130, Paris, France, October 28-30 2002.

[DMCF99] N. Davies, K. Mitchell, K. Cheverst, and A. Friday. Caches in the Air: Disseminating Tourist Information in the Guide System. In *Proceedings of IEEE Workshop on Mobile Computing Systems and Applications*, New Orleans, USA, September 1999.

[DRD+00] Alan Dix, Tom Rodden, Nigel Davies, Jonathan Trevor, Adrian Friday, and Kevin Palfreyman. Exploiting space and location as a design framework for interactive mobile systems. *ACM Transactions on Computer-Human Interaction*, 7(3):285–321, 2000.

[DS03] Anind K. Dey and Tim Sohn. Supporting End User Programming of Context-Aware Applications. In *Proceedings of Conference on Human Factors in Computing Systems (CHI), Workshop on Perspectives in End User Development*, Fort Lauderdale, FL, USA, April 5-10 2003.

[DSA99] Anind K. Dey, Daniel Salber, and Gregory D. Abowd. A Context-Based Infrastructure for Smart Environments. In *Proceedings of the 1st International Workshop on Managing Interactions in Smart Environments (MANSE)*, pages 114–128, Dublin, Ireland, December 1999.

[DSA01] Anind K. Dey, Daniel Salber, and Gregory D. Abowd. A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications. *Human-Computer Interaction (HCI) Journal*, 16:97–166, 2001.

[DSFA99] Anind K. Dey, Daniel Salber, Masayasu Futakawa, and Gregory D. Abowd. An Architecture to Support Context-Aware Applications. Technical Report GIT-GVU-99-23, Georgia Institute of Technology, June 1999.

[Duc92] D. Duchamp. Issues in wireless mobile computing. In *Third Workshop on Workstation Operating Systems*, pages 2–10, Key Biscayne, Florida, U.S., 1992. IEEE Computer Society Press.

[(Ed81a] Jon Postel (Ed.). RFC 791 - Internet Protocol, DARPA Internet Program, Protocol Specification, September 1981.

[(Ed81b] Jon Postel (Ed.). RFC 793 -Transmission Control Protocol (TCP), DARPA Internet Program, Protocol Specification, September 1981.

[Eng97] John English. *Ada 95: The Craft of Object-Oriented Programming.* Prentice Hall, 1997.

[Eug91] Eugene Charniak. Bayesian Networks without Tears. *Artificial Intelligence (AI) Magazine*, 12(4):50–63, 1991.

[FH03] Ernest Friedman-Hill. *JESS In Action, Rule-Based Systems in Java.* Manning, 2003.

[fS86] International Organization for Standardization. ISO 8879:1986. Information Processing - Text and Office Systems - Standard Generalized Markup Language (SGML), October 1986.

[FVB02] Alois Ferscha, Simon Vogl, and Wolfgang Beer. Ubiquitous Context Sensing in Wireless Environments. In *Proceedings of* 4^{th} *Austrian-Hungarian Workshop on Distributed and Parallel Systems (DAPSYS)*, Linz, Austria, October 2002. Kluwer Academic Publishers.

[FZ94] George H. Forman and John Zahorjan. The Challenges of Mobile Computing. *IEEE Computer*, 27(6):38–47, April 1994.

[GA94] Avelino J. Gonzalez and Robert Ahlers. A Novel Paradigm for Representing Tactical Knowledge in Intelligent Simulated Opponents. In *Proceedings of 7^{th} International Conference on Industrial Engineering Applications and A.I. and Expert Systems*, June 1994.

[GA95] Avelino J. Gonzalez and Robert Ahlers. Context-based Representation of Intelligent Behaviour in Simulated Opponents. In *Proceedings of 5^{th} Conference on Computer Generated Forces and Behavior Representation*, Orlando, Florida, May 1995.

[GA99] A. J. Gonzalez and R. Ahlers. Context-based Representation of Intelligent Behavior in Training Simulations. *Transactions of the Society for Computer Simulation International*, 15(4):153–166, March 1999.

[GBS00] Hans-W. Gellerson, Michael Beigl, and Albrecht Schmidt. Sensor-based Context-Awareness for Situated Computing. In *Proceedings of Workshop on Software Engineering and Pervasive Computing SEWPC00*, June 2000.

[Gov96] United States Government. Federal Standard 1037C - Glossary of Telecommunications Terms, August 7^{th} 1996.

[GPG00] Fernando G. Gonzalez, Grejs Patric, and Avelino J. Gonzalez. Autonomous Automobile Behaviour through Context-based Reasoning. In *Proceedings of 13^{th} Annual Florida Artificial Intelligence Research Society Conference*, Orlando, Florida, May 22-24 2000.

[Gro02] Object Management Group. The Common Object Request Broker: Architecture and Specification, V3.0, July 2002.

[GSB02] H-W. Gellersen, A. Schmidt, and M. Beigl. Multi-Sensor Context-Awareness

in Mobile Devices and Smart Artefacts. *ACM journal Mobile Networks and Applications (MONET)*, 7(5), October 2002.

[Haa03] Mads Haahr. *Supporting Mobile Computing in Object-Oriented Middleware Architectures*. Phd thesis, University of Dublin, Trinity College, October 2003.

[HB98] Tom Holvoet and Yolande Berbers. Composing Distributed Applications through Generative Communication. In *Proceedings of the 8th ACM SIGOPS European workshop on Support for Composing Distributed Applications*, pages 214–221, Sintra, Portugal, September 1998.

[Her03] Jack Herrington. *Code Generation in Action*. Manning, 2003.

[HH94] A. Harter and A. Hopper. A Distributed Location System for the Active Office. *IEEE Network*, 8(1):62–70, 1994.

[HHS+99] Andy Harter, Andy Hopper, Pete Steggles, Andy Ward, and Paul Webster. The Anatomy of a Context-Aware Application. In *Proceedings of the Fifth annual ACM/IEEE International Conference on Mobile Computing and Networking, MOBICOM '99, Seattle, Washington*, pages 59–68, August 1999.

[HIR02] Karen Henricksen, Jadwiga Indulska, and Andry Rakotonirainy. Modeling Context Information in Pervasive Computing Systems. In F. Mattern and M. Naghshineh, editors, *Proceedings of the First International Conference on Pervasive Computing*, number LNCS 2414, pages 167–180. Springer Verlag Berlin Heidelberg, 2002.

[HK97] Markus Horstmann and Mary Kirtland. DCOM Architecture. Microsoft White Paper, July 23 1997.

[HL01] J. I. Hong and J. A. Landay. An Infrastructure Approach to Context-Aware Computing. *Human-Computer Interaction*, 16(2-4):287–303, 2001.

[HL04] Jason I. Hong and James A. Landay. An Architecture for Privacy-Sensitive Uniquitous Computing. In *Proceedings of the* 2^{nd} *International Conference on*

Mobile Systems, Applications, and Services (Mobisys 2004), Boston, Ma., USA, June 6-9 2004.

[HM93] Jim C Hoffman and Robin R. Murphy. Comparison of bayesian and dempster-shafer theory for sensing: a practitioner's approach. *Proc. SPIE Neural and Stochastic Methods in Image and Signal Processing II*, 2032, July 1993.

[HNBR97] R. Hull, P. Neaves, and J. Bedford-Roberts. Towards Situated Computing. In *In Proceedings ISWC '97 (1st International Symposium on Wearable Computers)*, pages 146–153, Cambridge, MA, USA, October 13-14 1997.

[Hon00] Jason I. Hong. Context Fabric: Infrastructure Support for Context-Aware Systems. Qualifying Exam Proposal, Department of Computer Science, University of California at Berkeley, 2000.

[Hon02] J.I Hong. The Context Fabric: An Infrastructure for Context-Aware Computing. In *Proceedings of Doctoral Consortium, Human Factors in Computing Systems: CHI*, Minneapolis, USA, 2002.

[Hop00] Andy Hopper. The Royal Society Clifford Paterson Lecture, 1999 - Sentient Computing. *Phil. Trans. R. Soc. Lond.*, 358:2349–2358, August 2000.

[HV02] G. Holland and N. Vaidya. Analysis of TCP Performance over Mobile Ad Hoc Networks. *Wireless Networks*, 8(2-3):275–288, 2002.

[HWLC94] Hoffmann-Wellenhof, B. H. Lichtenegger, and J. Collins. *GPS: Theory and Practice*. Springer-Verlag, New-York, 3rd edition, 1994.

[IB92] T. Imielinski and B. R. Badrinath. Mobile wireless computing: Solutions and challenges in data management. Technical report, Department of Computer Science, Rutgers University, U.S., 1992.

[IRRH03] J. Indulska, R. Robinson, A. Rakotonirainy, and K. Henricksen. Experiences In Using CC/PP In Context-Aware Systems. In *Proceedings 4th International Con-*

ference on Mobile Data Management (MDM '03), pages 247–261, Melbourne, Australia, January 2003.

[Jac88] V. Jacobson. Congestion Avoidance and Control. In *Proceedings of ACM SIGCOMM*, pages 314–329. ACM, 1988.

[Jen02] C. D. Jensen. Secure Collaboration in Global Computing Systems. *European Research Consortium for Informatics and Mathematics (ERCIM) News*, 49, 2002.

[JM96] David B Johnson and David A Maltz. Dynamic source routing in ad hoc wireless networks. In Imielinski and Korth, editors, *Mobile Computing*, volume 353, pages 153–181. Kluwer Academic Publishers, 1996.

[JN04] J.P.Lewis and Ulrich Neumann. Performance of Java versus C++. [On-Line] Available: http://www.idiom.com/~zilla/Computer/javaCbenchmark.html, 2004.

[Kal60] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, 82(Series D):35–45, 1960.

[KCM+01] M. O. Killijian, R. Cunningham, R. Meier, L. Mazare, and V. Cahill. Towards Group Communication for Mobile Participants. In *Proceedings of ACM Workshop on Principles of Mobile Computing (POMC'2001)*, pages 75–82, Newport, Rhode Island, USA, 2001.

[KFJ01] L. Kagal, T. Finin, and A. Joshi. Trust-Based Security in Pervasive Computing Environments. *IEEE Computer*, 24(12):154–157, December 2001.

[KLM96] Leslie P. Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement Learning: A Survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[LA01] Kristof Van Laerhoven and Kofi A. Aidoo. Teaching Context to Applications. *Personal and Ubiquitous Computing*, 5(1):46–49, February 2001.

[Lae00] Kristof Van Laerhoven. TEA Project Homepage. `http://www.teco.edu/tea/`, November 2000.

[Lan66] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, May 1966.

[Lan02] Marc Langheinrich. Privacy Invasions in Ubiquitous Computing. In *Proceedings of Workshop on Socially-informed Design of Privacy-enhancing Solutions in Ubiquitous Computing, UbiComp 2002*, G$^{'}$oteborg, Sweden, September 2002.

[LAZC00] S. Lee, G. Ahn, X. Zhang, and A. Campbell. INSIGNIA: An IP-based quality of service framework for mobile ad hoc networks. *Journal of Parallel and Distributed Computing*, 60(4), April 2000.

[LBC+01] J. Li, C. Blake, D. S. J. D. Couto, H. I. Lee, and R. Morris. Capacity of ad hoc wireless networks. In *Proceedings 7th ACM International Conference Conference on Mobile Computing and Networking*, 2001.

[Lio90] Yihwa Irene Liou. Knowledge acquisition: issues, techniques, and methodology. In *Proceedings of the ACM SIGBDP conference on Trends and directions in expert systems*, pages 212 – 236, Orlando, FL, USA, 1990.

[LKAA96] Sue Long, Rob Kooper, Gregory D. Abowd, and Christopher G. Atkeson. Rapid Prototyping of Mobile Context-Aware Applications. In *Proceedings of the 2nd ACM International Conference on Mobile Computing and Networking, Mobicom*, pages 97–107, Rye, New York, USA, 1996.

[MC02a] R. Meier and V. Cahill. STEAM: Event-Based Middleware for Wireless Ad Hoc Networks. In *Proceedings of the International Workshop on Distributed Event-Based Systems (ICDCS/DEBS'02)*, pages 639–644, Vienna, Austria, 2002. IEEE Computer Society.

[MC02b] R. Meier and V. Cahill. STEAM: Event-Based Middleware for Wireless Ad Hoc Networks. In *Proceedings of the International Workshop on Distributed Event-*

Based Systems (ICDCS/DEBS'02), pages 639–644, Vienna, Austria, 2002. IEEE Computer Society.

[MC03] R. Meier and V. Cahill. Exploiting Proximity in Event-Based Middleware for Collaborative Mobile Applications. In *Proceedings of the* 4^{th} *IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS '03), LNCS 2893*, pages 285–296, Paris, France, 2003. Springer-Verlag.

[Mei03] René Meier. *Event-Based Middleware for Collaborative Ad Hoc Applications*. PhD thesis, University of Dublin, Trinity College, September 2003.

[Moc87] P. Mockapetris. RFC 1034 - Domain Names - Concepts and Facilities, November 1987.

[Moz98] M. C. Mozer. The neural network house: An environment that adapts to its inhabitants. In *Proccedings of the AAAI Spring Symposium on Intelligent Environments*, pages 110–114, 1998.

[MSB04] Eric Miller, Ralph Swick, and Dan Brickley. Resource Description Framework Homepage. `http://www.w3.org/RDF/`, 2004.

[Mun02] R. Muntz. MUSE Project Overview. `http://mmsl.cs.ucla.edu/muse`, 2002.

[NAS99] NASA. CLIPS: A Tool for building Expert Systems. `http://www.ghg.net/clips/CLIPS.html`, August 1999.

[NAU93] B. Clifford Neuman, Steven Seger Augart, and Shantaprasad Upasani. Using Prospero to support integrated location-independent computing. In *Proceedings USENIX Symposium on Mobile & Location-Independent Computing*, pages 29–34, August 1993.

[O'C04] Neil O'Connor. Sentient Car Hardware Write-up. Internal Report, Department of Computer Science, Trinity College Dublin, May 2004.

[OR02] Patrik Osbakk and Nick Ryan. Context, CC/PP, and P3P. In Peter Ljungstrand and Lars Erik Holmquist, editors, *Adjunct Proceedings, UbiComp 2002*, pages 9–10, Goteborg, Sweden, September 2002.

[OR03] Patrik Osbakk and Nick Ryan. A Privacy Enhancing Infrastructure for Context-Awareness. In *Proceedings of* $1^{st}UK-UbiNetWorkshop$, Imperial College, London, UK, September 2003.

[Pas98] J. Pascoe. Adding Generic Contextual Capabilities to Wearable Computers. In *Proceedings of* 2^{nd} *International Symposium on Wearable Computers*, pages 92–99. IEEE CS Press, Los Alamitos, California, 1998.

[Per97] Charles E. Perkins. Mobile IP. *IEEE Communications Magazine*, 35(5):84–99, May 1997.

[PMR98] J. Pascoe, D. R. Morse, and N. S. Ryan. Developing Personal Technology for the Field. *Personal Technologies*, 2(1):28–36, 1998.

[PR99] Charles E. Perkins and Elizabeth M. Royer. Ad hoc On-Demand Distance Vector Routing. In *Proceedings of the* 2^{nd} *IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100, New Orleans, LA, USA, February 1999.

[PR03] Judea Pearl and Stuart Russell. *Michael A. Arbib (Ed.) Bayesian Networks, The Handbook of Brain Theory and Neural Networks.* MIT Press, 2^{nd} edition, 2003.

[PRB98] J. Pascoe, N. S. Ryan, and P. J. Brown. Context aware: the dawn of sentient computing? *GPS World*, 9(9):22–30, September 1998.

[PRJ04] J. Payton, G.-C. Roman, and C. Julien. Context Sensitive Data Structures Supporting Software Development in Ad Hoc Mobile Settings. In *Proceedings of the* 3^{rd} *International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS'2004)*, pages 34–41, Edinburgh, Scotland, UK, May 2004.

[PRM98] J. Pascoe, N. S. Ryan, and D. R. Morse. Human Computer Giraffe Interaction: HCI in the Field. In *Proceedings of Workshop on Human Computer Interaction with Mobile Devices*, University of Glasgow, United Kingdom, May 21-23 1998.

[PRM99] Jason Pascoe, Nick Ryan, and David Morse. Issues in Developing Context-Aware Computing. In H.-W. Gellerson, editor, *Proceedings of Handheld and Ubiquitous Computing: First International Symposium, HUC'99*, Lecture Notes in Computer Science 1707, pages 208–221, Karlsruhe, Germany, September 1999. Springer Verlag Berlin Heidelburg.

[RAMC04] Anand Ranganathan, Jalal Al-Muhtadi, and Roy H. Campbell. Reasoning about Uncertain Contexts in Pervasive Computing Environments. *IEEE Pervasive Computing*, 3(2), April-June 2004.

[RB01] P. Reinbold and O. Bonaventure. A Comparison of IP Mobility Protocols. Technical Report Infonet-TR-2001-07, University of Namur, June 2001.

[RC00] Manuel Román and Roy H. Campbell. GAIA: Enabling Active Spaces. In *Proceedings of 9^{th} ACM SIGOPS European Workshop*, pages 229–234, Kolding, Denmark, September 17-20 2000.

[RC02] Manuel Román and Roy H. Campbell. A User-Centric, Resource-Aware, Context-Sensitive, Multi-Device Application Framework for Ubiquitous Computing Environments. Technical Report UIUCDCS-R-2002-2284 UILU-ENG-2002-1728, University of Illinois at Urbana-Champaign, 2002.

[RC03a] Anand Ranganathan and Roy H. Campbell. A Middleware for Context-Aware Agents in Ubiquitous Computing Environments. In *Proceedings of ACM/IFIP/USENIX International Middleware Conference*, Rio de Janeiro, Brazil, June 16-20 2003.

[RC03b] Anand Ranganathan and Roy H. Campbell. An infrastructure for context-awareness based on first order logic. *Personal and Ubiquitous Computing*, 7(6):353–364, December 2003.

[RHC+02] Manuel Román, Christopher K. Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell, and Klara Nahrstedt. Gaia: A Middleware Infrastructure to Enable Active Spaces. *IEEE Pervasive Computing*, 1(4):74–83, Oct-Dec 2002.

[Rho97] B.J. Rhodes. The wearable remembrance agent: a system for augmented memory. In *Proceedings of* 1^{st} *International Symposium on Wearable Computers*, pages 123–128, Cambridge, Massachusetts, USA, October 1997.

[RJD99] N.S. Ryan, J.Pascoe, and D.R.Morse. FieldNote: a Handheld Information System for the Field. In R.Laurini, editor, *Proc. TeleGeo'99,* 1^{st} *International Workshop on TeleGeoProcessing*, pages 156–163. Claude Bernard University of Lyon, May 1999.

[RMCM03] Anand Ranganathan, Robert E. McGrath, Roy H. Campbell, and M. Dennis Mickunas. Ontologies in a Pervasive Computing Environment. In *Proceedings of Workshop on Ontologies and Distributed Systems (part of the* 18^{th} *International Joint Conference on Artificial Intelligence IJCAI 2003)*, Acapulco, Mexico, August 9 2003.

[RPM97] N. Ryan, J. Pascoe, and D. Morse. Enhanced Reality Fieldwork: the Context-Aware Archaeological Assistant. In: Computer Applications in Archaeology, Gaffney, V., van Leusen, M., Exxon, S. (eds.), 1997.

[Rya99] Nick Ryan. ConteXtML: Exchanging Contextual Information between a Mobile Client and the FieldNote Server, August 1999.

[RZC03] Manuel Román, Brian Ziebart, and Roy Campbell. Dynamic Application Composition: Customizing the Behavior of an Active Space. In *Proceedings of IEEE International Conference on Pervasive Computing and Communications (PerCom 2003)*, Dallas-Fort Worth, Texas, USA, March 23-26 2003.

[Saf97] Paul Saffo. Sensors: The next wave of infotech innovation. On-Line: Available `http://www.saffo.com/sensors.html`, 1997.

[Sat96] M. Satyanarayanan. Fundamental challenges in mobile computing. In *Symposium on Principles of Distributed Computing*, pages 1–7, 1996.

[SAT+99] Albrecht Schmidt, Kofi Asante Aidoo, Antti Takaluoma, Urpo Tuomela, Kristof Van Laerhoven, and Walter Van de Velde. Advanced interaction in context. In H.W. Gellersen, editor, *Proceedings of First International Symposium on Handheld and Ubiquitous Computing (HUC99)*, volume 1707 of LNCS, pages 89–101. Springer-Verlag, September 1999.

[Sat01] M. Satyanarayanan. Pervasive Computing: Vision and Challenges. *IEEE Personal Communications*, 8(4):10–17, August 2001.

[SAW94] Bill Schilit, Norman Adams, and Roy Want. Context-Aware Computing Applications. In *Proceedings of 1st IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, US, December 8-9 1994.

[SB98] A. Schmidt and M. Beigl. There is More to Context than Location. In *Proceedings of the International Workshop on Interactive Applications of Mobile Computing*, Rostock, Germany, November 1998.

[SDA99] Daniel Salber, Anind K. Dey, and Gregory D. Abowd. The Context Toolkit: Aiding the Development of Context-Enabled Applications. In *Proceedings of the 1999 Conference on Human Factors in Computing Systems (CHI), Pittsburgh, PA*, pages 434–441, May 1999.

[SF99] Albrecht Schmidt and Jessica Forbess. What GPS Doesn't Tell You: Determining One's Context with Low-Level Sensors. In *Proceedings of the 6th IEEE International Conference on Electronics, Circuits and Systems*, Paphos, Cyprus, September 5-8 1999.

[Shi89] Paul A. Shirley. An Introduction to Ultrasonic Sensing. *Sensors - The Journal of Machine Perception*, 6(11), November 1989.

[SSF+03] D. Siewiorek, A. Smailagic, J. Furukawa, N. Moraveji, K. Reiger, and J. Shaffer. SenSay: A Context-Aware Mobile Phone. In *Proceedings of IEEE International Symposium on Wearable Computers*, New York, NY, USA., 2003.

[SSL+02] Albrecht Schmidt, Martin Strohbach, Kristof Van Laerhoven, Adrian Friday, and Hans-W. Gellersen. Context Acquisition based on Load Sensing. In G. Boriello and Lecture Notes in Computer Science L.E. Holmquist (Eds)., editors, *Proceedings of Ubicomp 2002*, volume 2498, pages 333 – 351, Goteborg, Sweden, September 2002. Springer Verlag.

[ST94] Bill N. Schilit and Marvin M. Theimer. Disseminating Active Map Information to Mobile Hosts. *IEEE Network*, 8(5):22–32, September/October 1994.

[Ste00] Jeff Stefan. Navigating with GPS. *Circuit Cellar, Issue 123*, October 2000.

[STM00] Albrecht Schmidt, Antti Takaluoma, and Jani Mantyjarvi. Context-Aware Telephony Over WAP. *Personal Technologies*, 4(4):225–229, September 2000.

[STW93] B. Schilit, M. Theimer, and B. Welch. Customising mobile applications. In *Proceedings of USENIX Symposium on Mobile and Location-Independent Computing*, Cambridge, MA, USA, August 1993.

[Tan96] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall, Inc., third edition, 1996.

[TK01] Hans-Rolf Trankler and Olfa Kanoun. Recent Advances in Sensor Technology. In *Proceedings of* 18^{th} *IEEE Instrumentation and Measurement Technology Conference*, Budapest, Hungary, May 21-23 2001.

[Unk03] Unknown. The sentient office is coming. *Economist Technology Quarterly*, 367(8329):27, June 19^{th} 2003.

[vDKV00] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[Vis99] Arnoud Visser. Design and Organisation of Autonomous Systems. On-Line: Available `http://www.science.uva.nl/ arnoud/education/OOAS/`, August 1999.

[Wei91] Mark Weiser. The Computer for the 21^{st} Century. *Scientific American*, 265(3):94–104, September 1991.

[Wei93] M. Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):74–84, October 1993.

[WHFG92] R. Want, A. Hopper, V. Falcao, and J. Gibbons. The Active Badge Location System. *ACM Transactions on Information Systems*, 10(1):91–102, January 1992.

[Wil98] Svein Yngvar Willassen. A method for implementing mobile station location in gsm. Ms thesis, Department of Telematics, Norwegian University of Technology and Science, 1998.

[Win01] Terry Winograd. Architectures for Context. *Human-Computer Interaction (HCI) Journal*, 16(2-3), 2001.

[WJH97] Andy Ward, Alan Jones, and Andy Hopper. A New Location Technique for the Active Office. *IEEE Personal Communications*, 4(5):42–47, October 1997.

[Wol03] Muiris Wolfe. Smart Couch Report. Department of Computer Science, Trinity College Dublin, 2003.

[WRW96] A. Wollrath, R. Riggs, and J. Waldo. A Distributed Object Model for the Java System. In 2^{nd} *Conference on Object-Oriented Technologies & Systems (COOTS)*, pages 219–232. USENIX Association, 1996.

[WSA+95] R. Want, W. Schilit, N. Adams, R. Gold, K. Petersen, D. Goldberg, J. Ellis, and M Weiser. An Overview of the PARCTAB Ubiquitous Computing Environment. *IEEE Personal Communications*, 2(6):28–43, December 1995.

[WSA02] Huadong Wu, Mel Siegel, and Sevim Ablay. Sensor Fusion for Context Understanding. In *Proceedings of IEEE Instrumentation and Measurement Technology Conference (IMTC)*, Anchorage, AK, USA, May 2002.

[WSSY02] Huadong Wu, Mel Siegel, Rainer Stiefelhagen, and Jie Yang. Sensor Fusion Using Dempster-Shaefer Theory. In *Proceedings of IEEE Instrumentation and Measurement Technology Conference (IMTC)*, Anchorage, AK, USA, May 2002.

[Wu03] Huadong Wu. *Sensor Data Fusion for Context-Aware Computing Using Dempster-Shafer Theory*. PhD thesis, The Robotics Institute, Carnegie Mellon University, December 2003.

[XTB+03] K. Xu, K. Tang, R. Bagrodia, M. Gerla, and M. Bereschinsky. Adaptive Bandwidth Management and QoS Provisioning in Large Scale Ad Hoc Networks. In *Proceedings of IEEE Military Communications Conference (MILCOM)*, Boston, MA, USA, October 2003.

[ZR97] H. Ziv and D. Richardson. Bayesian-network confirmation of software testing uncertainties. Technical report, University of California, Irvine, 1997.