Extending Event Based Programming for Sensor-Driven Applications in Resource-Constrained Environments

Sean Reilly

A thesis submitted to the University of Dublin, Trinity College

in fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

September 2011

Declaration

I declare that this thesis has not been submitted as an exercise for a degree at this or any other university and it is entirely my own work.

I agree to deposit this thesis in the University's open access institutional repository or allow the library to do so on my behalf, subject to Irish Copyright Legislation and Trinity College Library conditions of use and acknowledgement.

Sean Reilly

Dated: 27th September 2011

Acknowledgements

Firstly, I would like to thank my supervisor Dr. Mads Haahr for his guidance and encouragement. I have learned a great deal from working with him and appreciate the enormous contribution he has made to my academic development.

I would also like to thank my colleagues and friends in the Distributed Systems Group for making it a pleasure to be part of this unique research environment.

Thank you to all my friends for their help and encouragement along the way and for making my time as a student as fulfilling and rewarding as it was.

Finally, I would like to especially thank my parents and family for their constant support for all of my academic endeavours over the years. Without you this simply would not have been possible.

Sean Reilly
University of Dublin, Trinity College
September 2011

Summary

The ubiquitous computing vision which began with Weiser and its evolution since predicts large quantities of sensors, actuators and computational devices embedded in our environment enabling advanced applications which increase the ambient intelligence of the environment. As the prevalence of these devices increase, this vision is moving closer to reality. The large quantities of sensors have the potential to enable a range of sensor-rich applications in diverse domains such as healthcare, computer augmented sports, intelligent homes and smart offices. Within this class of sensor-rich applications are two subsets, sensor monitoring applications and sensor-driven applications. Sensor monitoring applications are applications that monitor sensors and transmit data to an external sink. The second sub-set of these sensor-rich applications which we call sensor-driven applications, i.e. applications that change their behaviour in response to sensor input, will provide the main instantiation of the ubiquitous computing vision. Sensor driven applications have two defining characteristics. Firstly they have multiple sensors and secondly they need to combine the data from these multiple sensors in order to drive the behaviour of the application.

There are a large amount of challenges in pervasive computing, ranging from privacy, trust, heterogeneous devices and real-time applications to resource usage, routing, reliability and fault tolerance. In this thesis we focus on programming challenges which must be overcome when developing ubiquitous computing applications. Despite the fact that there are so many different types of applications a large amount of these applications share a common challenge, which is the need to deal with multiple sensor streams. Two challenges related to dealing with multiple sensor streams which we address in this thesis are the complexity of combining multiple sensor streams and the challenge of dealing with large quantities of continuous data. As the number of sensors in the system increases the developer must find a way to combine the various streams of sensor data in order to perform meaningful multi-sensor data fusion and drive the behavior of the application. The continuous nature of sensor data also provides challenges to the developer.

Previous work in this area is split in two main categories, event processing and stream processing. Event based programming is popular for developing a large range of applications but has been found to be particularly useful for developing ubiquitous computing applications. The asynchronous nature of the communication, the decoupled development and the fact that it is relatively easy to map sensed phenomena to events has led to event based programming being a standard approach to developing ubiquitous computing applications.

Stream processing is an area that ranges from stream oriented programming languages in which streams are a first class type to large middleware systems devoted to the manipulation of streams. These large systems are used to perform continuous queries over streams of data in an analogous way to traditional database queries. Stream processing provides facilities for dealing with large quantities of sensor data and performing queries and operations across this data. However despite providing abstractions which could prove beneficial to developers stream processing has largely not yet been adopted by ubiquitous computing developers.

Both of these solutions to the challenges of developing applications in the domain of sensor-driven applications have their own individual merits. Event based programming is a well proven tool of developers of ubiquitous computing applications while stream processing has specialised abstractions for combining streams and handling continuous data. In this thesis we propose a novel model that combines elements from stream processing with event based programming in order to make it more suitable for developing sensor-driven applications.

This thesis provides a set of abstractions which are used to extend event based programming to make it more suitable for developing sensor-driven applications. The combined model, Architecture for fusing events using streams and execution policies (AESOP) makes it easier for programmers to combine sensor streams by providing multi-event handlers and execution policies. AESOP also provides event streams which are abstractions that reduce the burden on the developer caused by programming with continuous data. The large range of ubiquitous computing platforms places additional challenges on any abstractions that aim to be applicable in the domain. As a large number of these platforms are resource-constrained any general solution must also be suitable for resource-constrained platforms. The AESOP extensions are also designed to be useful with any underlying event model, be it peer-to-peer, implicit or mediator.

To evaluate the approach, we examine the generality of the abstractions and their suitability for writing sensor-driven applications. To evaluate our abstractions we have implemented two instantiations of the AESOP model. The first, C-AESOP is written in C on the Gumstix platform. It is implemented on top of the STEAM event model. The second implementation, J-AESOP is written in Java on the Android platform and uses an event model based on the event listener pattern. The abstractions are shown to be general across programming languages, event models and applications. They are also shown to be suitable for solving the challenges of combining sensor streams and dealing with continuous data and also that the abstractions are suitable for use on resource constrained platforms.

Publications Related to this Ph.D.

- Sean Reilly, Mads Haahr, *"Extending the event-based programming model to support sensor-driven ubiquitous computing applications",* Pervasive Computing and Communications, IEEE International Conference on, pp. 1-6, 2009 IEEE International Conference on Pervasive Computing and Communications, 2009.

- Sean Reilly, *"Multi-Event Handlers for sensor-driven ubiquitous computing applications",* Pervasive Computing and Communications, IEEE International Conference on, pp. 1-2, 2009 IEEE International Conference on Pervasive Computing and Communications, 2009.

- Reilly, S., Barron, P., Cahill, V., Moran, K., and Haahr, M. (2009). *A General-Purpose Taxonomy of Computer-Augmented Sports Systems.* In Nigel K. Ll. Pope, Kerri-Ann L. Kuhn and John Forster (Eds.), Digital Sport for Performance Enhancement and Competitive Evolution: Intelligent Gaming Technologies. Hershey, PA: IGI Global.

Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

The ubiquitous computing vision initially proposed by Weiser [136] and developed since is one in which many sensors, actuators and computational devices are distributed all around us in the environment in which we live. This vision is becoming ever close to reality with many people possessing multiple computational consumer devices, e.g., desktop computers, mobile phones [106] and in-car computers but also with advances in wireless sensor networks and embedded devices [132]. Developing ubiquitous computing applications for these devices, however, has numerous differences to developing traditional desktop applications. For example, problems of mobility, heterogeneous devices, privacy and sensing the environment are all much more acute and complex in ubiquitous computing applications. In this thesis we propose abstractions that are suitable for combining and analysing multiple sensor streams and in doing so make it easier to develop applications which use multiple sensors. The abstractions are presented as generic extensions to event based systems. We present two implementations of the abstractions which we use in our evaluation of the abstractions.

This introductory chapter motivates the body of work presented in the thesis and provides background information in event-based programming and stream processing which is required to understand the thesis. The chapter also summarises the contribution of the thesis and presents a document roadmap.

1.1 Ubiquitous Computing and Sensor-Driven Applications

From when it was first proposed by Mark Weiser in his seminal 1991 paper, ubiquitous computing has developed into a research field in its own right. Weiser began with a vision of pervasive computing that was largely office and workspace based. This initial approach has since been expanded upon by many researchers. There are now a large amount of visions of ubiquitous computing from smart homes [83], smart dust [96], ambient intelligence [116], pervasive healthcare [84], personal information auras [48] to people-centric sensing [22]. These visions pay testament to the fact that there is a very broad spectrum of ubiquitous computing applications from tiny embedded systems performing habitat monitoring [96] through to large scale systems to support users in their residence [83].

Despite the fact that ubiquitous computing now is driven by a large number of different approaches and is used to describe a broad variety of applications, one characteristic which is common to a large amount of these applications is that of having one or more sensors and needing to combine the streams of data from these sensors in order to react and respond to their environment. We use the term *sensor-rich* applications to describe the class of applications that have multiple sensors. We define *sensor-driven* applications as a sub-class of sensor-rich applications. Sensor-driven applications have two main defining characteristics: they have multiple sensors, and they change their behaviour primarily in response to their sensor inputs. This in contrast to applications which might perform some monitoring tasks using sensors without any combination or analysis. An example of a sensor-driven application would be a pool monitoring system such as Poseidon [75] which uses multiple sensors to detect incapacitated swimmers and alert lifeguards to their whereabouts. Sensor-driven applications are a very significant sub-set of all ubiquitous computing applications.

1.2 Event-Based Programming

Event-based programming is a very popular paradigm for building computer systems. It has been used to build a wide range of systems from graphical user interfaces [121] to massively distributed banking systems [5] and deployed across the spectrum of platforms from tiny motes [59] to the Grid [80]. Event-based system use *events* to communicate between individual components of the system. Each event has an event source and can be delivered to multiple consumers of the event. Events can carry some relevant information e.g., a timestamp signifying when the event happened. An *event model* is a specification of the event and a description of the infrastructure for communicating each event. The characteristics of the event model are perhaps best thought about in terms of communication. In the traditional client/server approach one client interacts with one server. The communication is generally one-to-one. This is in contrast with the event model where there is generally more than one recipient of the event. Also in the client/server model, the client usually (though not always) expects a response from the server. In the event model, the communication is usually asynchronous and often anonymous, i.e., the communicating elements do not necessarily need to know about the receivers or producers of their events, just that an event has happened. This allows decoupling of the producer of the events from the consumer. In the event model, all parties are permitted to initiate communication, i.e., send events and to receive communications, i.e., receive events. This is in contrast to the client/server model where the client always initiates communication with the server. The asynchronous one-to-many communication and the ability to decouple and modularise the components of the application make the event model a tool of choice for developers and designers of distributed systems. The event-based model is particularly suitable for writing applications that use sensors, because each individual sensor reading can be modelled directly as an event and sensor readings can be synchronous or asynchronous, both of which are supported by the event model. Ubiquitous computing systems also tend to be distributed and significant energy savings can be gained from using the event model, as the application only reacts to events that have occurred and

so can put the processor in idle mode between events. For example, the leading embedded operating system TinyOS [59] for Berkeley motes uses an event-based model. For these reasons the paradigm of event-based programming is particularly well suited to, and has been extensively used in, the development of ubiquitous computing applications [61].

There are a large variety of different event-based systems [112, 52, 103, 129, 99, 101, 110, 92], some designed for use on one machine and others distributed across many. There are also many types of event models including peer-to-peer [104], mediator [52] and implicit event models [15, 101]. There is a history of extending these event models and systems to make them better for dealing with a particular problem or for working in a particular domain. For example, scalability is a major concern in distributed systems and several extensions to the event-model have been proposed improve the scalability of applications that use event-based communication. Propagating events to all nodes in a large distributed system severely hampers the scalability of the system. To this end work like the ECO [110] model and Cambridge Event Model [112] introduce mechanisms to limit the distribution of irrelevant or unneeded events. In the ECO model the mechanisms are called *notify constraints* and in the Cambridge Event Model these are referred to as *filters*. This is just one example of a group of extensions to the core event model designed to make it more suitable for dealing with a particular challenge or application domain.

1.3 Stream Processing

Stream Processing as a research discipline has its origins in stream processing languages such as Lucid [11] from the late 1970s. In stream processing languages, streams are a first order data type. These languages are usually dataflow languages, i.e., systems in which the movement of data around the system causes the program to execute. For example, in Lucid the expression $a + b = c$ will wait until it has a value for a and for b and then evaluate the value for c. Stream processing languages have found favour with functional programming researchers because of their strong semantics and formal underpinnings. They have also been used in hardware design languages and other fields.

Streams, or more specifically *data streams* are time ordered sequences of data tuples. There is a strict time ordering on the elements of the data stream. Each individual tuple in a data stream contains some stream specific information, but can also contain additional information such as a timestamp which can be used in the processing of the stream.

Since the late 1990s focus is returning to stream processing but with a different set of goals. The large quantities of data available to computers from sources as diverse as stock exchanges, sensors and RSS feeds and the ubiquity of networking connections mean that computers have access to large volumes of data that need to be processed in order to extract relevant information and detect interesting trends. The standard approach to designing systems that needed to analyse large quantities of data (e.g. shop inventory systems) was to store this data in a relational database and then query the data at a later date. Because these data streams have such large amounts of data it is infeasible to store all the data and later process it so the data needs to be processed on the fly. Traditional

relational databases systems are insufficient for the task. Attempts have been made to extend relational databases to make them more suitable, for example by designing in-memory databases where the entire database resides in main memory to speed up the operations, however these efforts have largely failed due to the large volumes of data and it became clear that a new approach was required. This research has progressed to the development of data stream management systems (DSMS) which are specifically designed to deal with continuous streams of data. These DSMSs usually provide the facility to answer long running continuous queries (CQs) over some combination of input streams. Large amount of research in current DSMSs is focused on dealing with bursty, or infrequent streams, quality of service (QoS), calculating approximate results to queries, windowing mechanisms, load shedding [82, 14, 98, 91] and scheduling [12, 24, 125].

The dataflow model is ideal for working with streams of data because in essence the arrival of data is what drives the flow of the application. This is similar in a large number of respects to event-driven systems where the arrival of events causes the execution of handlers and drives the execution of the application.

1.4 This Thesis

The research hypothesis of this work is that there exists abstractions which simplify the design and implementation of sensor-driven applications and that the abstractions presented in chapter 4 of this thesis are such abstractions.

It is clear that event-based programming is a useful paradigm for developing ubiquitous computing applications. However, developing applications with multiple sensor streams is still non-trivial for the programmer. Combining and analysing continuous streams of sensor data requires data structures and algorithms that are not common in typical desktop or server applications. Stream processing, on the other hand, has abstractions for dealing with and combining streams and handling continuous data. However, it has very low uptake among application developers in general and among ubiquitous computing application developers in particular. This thesis aims to combine the best of both worlds by marrying concepts from stream processing with the preferred development techniques for ubiquitous computing. The approach is to extend the event-based programming model with abstractions from the domain of stream processing to make it more suitable for developing sensor-driven ubiquitous computing applications.

The main contributions of this thesis are:

1. Generic extension to event-based programming to support the development of sensor-driven applications
2. Design and implementation of C-AESOP
3. Design and implementation of J-AESOP
4. Design and implementation of the Squash Training Application

5. Design and implementation of the Viking Ghost Hunt Application

The first contribution is the AESOP abstractions themselves. These abstractions are presented in Chapter 3. They are a generic extension to event based programming and the most important of all the contributions. Contributions 2 and 3 relate to the implementation of the AESOP abstractions. It is highly illustrative to see how the abstractions can be implemented and to see how a host event system can be extended with the AESOP abstractions. The fourth and fifth contribution show applications that have been developed using the AESOP abstractions. These applications show how the abstractions can be used and are used to evaluate the usefulness of the AESOP abstractions in the development of sensor-driven applications.

The research questions which motivated the work in this thesis are:

1. Can we devise a set of abstractions that would be useful for developing sensor-driven applications?

2. How should we implement these abstractions?

3. Do the AESOP abstractions help in the creation of sensor-driven applications?

The research questions are directly related to the contributions of this thesis. The first research question "Can we devise a set of abstractions that would be useful for developing sensor-driven applications?" is directly addressed by the first contribution which is a "Generic extension to event-based programming to support the development of sensor-driven applications". The second and third contributions which detail the C-AESOP and J-AESOP design and implementation are directly related to the second research questions which asks how the abstractions proposed as an answer to question one might be implemented. Finally the fourth and fifth contributions which provide example applications developed using the abstractions are used to answer the third research question and to show that indeed the abstractions are useful for the development of sensor driven applications.

The extension, called Architecture for fusing Events using Streams and Execution Policies (AESOP), provides abstractions for combining sensor data from multiple continuous sensors. The support provided by these abstractions for developing sensor-driven applications is evaluated. The evaluation is carried out by creating two instantiations of the AESOP model. These instantiations are then used to develop two sensor-driven applications. These applications and the two instantiations are then analysed and the degree of support offered by the AESOP model in the development of sensor-driven applications is evaluated. The resource usage of the instantiations are also quantified and shown to be acceptable for the development of sensor-driven applications.

1.5 Document Roadmap

The structure of this thesis is as follows.

Chapter 2 defines sensor-driven applications and derives requirements for abstractions to support the development of them.

Chapter 3 presents a survey of the state of the art of abstractions used to develop sensor-rich applications.

Chapter 4 presents the proposed extensions to event-based programming, called AESOP.

Chapter 5 describes two instantiations of the model. C-AESOP is written in C on the Gumstix platform while J-AESOP is written in Java on the Android platform.

Chapter 6 evaluates the model by examining the generality of the extensions and the suitability of the model for writing sensor-driven applications.

Chapter 7 presents conclusions and outlines future research directions.

Chapter 2

Sensor-Driven Applications

The purpose of this chapter is to derive the requirements for abstractions that support sensor-driven applications. This chapter presents four example sensor-driven applications from four separate application domains within the greater field of ubiquitous computing. The qualities of these applications are analysed to reveal the common characteristics of sensor-driven applications. These characteristics are then used to synthesise the requirements for middleware to support sensor-driven applications. There are seven requirements of sensor-driven applications and we divide these requirements into functional requirements and general middleware requirements.

2.1 Example Applications

In order to focus our discussion of sensor-driven applications, we present here four example application from different application domains. The applications presented serve as examples when discussing sensor-driven applications in the remainder of the thesis, and two of the applications are implemented as demonstrator applications and used as the basis for our evaluation in chapter 6.

2.1.1 Squash Training Application

There is a growing trend towards the augmentation of sports with computational and sensing ability in order to referee, ensure athlete safety, entertain spectators and train athletes [115]. The sport we have chosen to computationally augment is the sport of *squash rackets*. Squash rackets, or as it is commonly known squash, is a racket sport and so it is typical of a large class of sports where athletes use some form of artefact to make contact with a ball. Examples of other sports that exhibit this behaviour are golf, hockey, tennis and cricket. Therefore an application that successfully augments the sport of squash should be applicable to a wider group of sports. The existence of an artefact in the sport is also desirable from the point of view of augmenting the sport with computational ability as embedding sensors and computational devices in an artefact is preferable to attaching them to the athlete or embedding them in the environment as attaching computational devices to an athlete can be uncomfortable for the athlete and inconvenient, especially if they need to be precisely attached

by a technician. Squash courts are also situated indoors which aids development and testing of the application. Furthermore considerable squash expertise and numerous athletes were available locally by virtue of the author's contacts in the Dublin University Squash Rackets Club.

Squash is a racket sport played between two competing athletes. In the sport of squash the swing of the athlete is of vital importance. Athletes and trainers are interested in several characteristics of the swing and the stance of the athlete as they make contact with the ball. Accelerometers and orientation sensors placed on the athlete's racket could be used to detect the moment of contact with the ball and the orientation of the racket on impact. They could also be used to detect the start and end of the athlete's swing. Force sensors in the athlete's shoes could detect the stance of the athlete. The balance and movement of the athlete is of vital importance as they strike the ball. The proposed application could analyse the movement and swing of the athlete and could offer advice to the athlete and coach on strategies for improving the swing.

2.1.2 Elderly Patient Fall Detection

Much consideration has been given to how ubiquitous computing can help cater for the needs of sick and elderly people [84, 135, 134, 107, 81]. This research field has become known as ubiquitous healthcare or pervasive healthcare. Assisted living is a domain which has the goals of allowing elderly and disabled patients to live on their own in their own home. This has several important advantages such as reducing the need for expensive hospitalisation, reducing the chances that the patient will contract a secondary infection in hospital and supporting the independence of the patient by allowing them to be in control of their own condition. The following example ubiquitous computing application is used to support a patient in their day to day life in and around their own home.

An elderly person lives alone and is concerned that they might fall and be unable to contact emergency services or friends for help. An automatic fall monitoring system is proposed to detect a potential fall and alert the relevant people. The system needs to distinguish between resting (such as sitting and lying down) and a more violent fall. It uses multiple accelerometers distributed around the patient's body, one on the hip, on the wrist and embedded in the patient's shoe. The system operates by detecting if the patient's posture has changed from being upright to horizontal, when this is detected it then performs more complex analysis over the historical accelerometer data from the three sensors to determine if the change was caused by a fall or the patient making a voluntary movement. If a fall is detected the application alerts the patient and unless the alert is cancelled, calls for help.

2.1.3 Car Skid Control Application

New automobiles contain large amounts of computers and sensors. For example, there are over 35 unique electronic features in BMW automobiles which include sensors and computational elements [4]. These include features such as an automatic Lane Departure Warning which analyses video streams of the oncoming road to detect when the automobile is drifting from its lane on a motorway at

high speed; Dynamic Stability Control which uses a multitude of sensors monitoring wheel rotation, steering angle, lateral forces, pressure and yaw to detect anomalies in the motion of the car and then initiates actions to correct these anomalies; and Active Safety system which uses sensors distributed around the car to detect the direction and force of impacts and uses this information to control airbags, active headrests and seat belt systems.

We propose an example application that is a skid control and recovery system designed for an automobile. The system has multiple sensors detecting the rate of turn of each of the cars wheels. It also has a momentum sensor which is a combination of several other sensors and also an accelerometer sensor. A skid is detected from analysing the rate of turn of all the wheels. When a skid is detected, analysis is performed using the momentum of the car, the accelerometer data and wheel turn rate for the period up to and including the skid and the best corrective action is calculated in order to take the car out of the skid and regain control of the car.

2.1.4 User Interface for Smartphone

Phones are increasing in capabilities both in terms of additional GPS, accelerometers and other sensors, as well as processing and graphics power. The iPhone [65], HTC G1 [38] and Nokia N96 [109] all support accelerometers and increased processing power in comparison to older models. The features available and the availability of a marketplace and central distribution system has led to a large amount of applications being developed for these devices. For example, the iPhone app store has had over two billion applications downloaded as of September 2009 [64] and has over one hundred thousand applications available as of November 2009 [63]. The largest category of applications available on the iPhone app store is in the game category. Clearly there is a large interest in developing and playing games for smartphones, however developing user interfaces with these sensors is not trivial as developers generally do not have the expertise to deal with sensors and programming in the physical world. One category of games which are well placed to utilise the advanced sensor functionality of the smartphones are location-aware games. Location-aware games are games in which the location of the player is used as an integral part of the game play, implying that the player can move around while playing the game. With the integration of GPS in most of the new smart phones the scene is set for location-aware games which allow the phone to be used in a non-traditional manner as the interface to the game.

The example application proposed is a location-aware game which is based on the genre of ghost stories, and paranormal investigation. In the game the player uses their phone as a paranormal investigation device. The player can peer through the phone and see ghosts overlaid on the images from the phones video feed. The movement of the ghost must match the movement of the phone, i.e. panning of the phone must allow the phone and imagery to move but the ghost should stay stationary in the scene.There are several sensors in the phone including magnetometers, accelerometers and GPS and these are combined and analysed in order to drive the game behaviour and create a novel interaction technique for the game.

2.2 Application Characteristics

When we consider the sensor-driven applications described in section 2.1 a number of characteristics related to the use of sensors which are common to all of the applications become apparent.

2.2.1 Multiple Sensors

Of the four applications it is clear that all posses multiple sensors. The squash application has four distinct sensors: an accelerometer, an orientation sensor and two separate force sensors. The automatic skid control application has six separate sensors: four wheel turn sensors, one momentum sensor and an accelerometer. The fall detection system has three separate accelerometer sensors and the user interface for a smartphone has three separate sensors: a magnetometer, an accelerometer and a GPS sensor. From this brief analysis of the sensor units in the example applications it is clear that a common characteristic of sensor-driven applications is the presence of multiple sensors.

2.2.2 Combination of Sensor Data

In section 1.1 we defined sensor-driven applications as applications which did not just store or forward sensor data but used this sensor data to drive the behaviour of the application. As observed in section 2.1 each sensor-driven application has multiple sensors. Therefore it follows that the sensor streams from the multiple sensors must be combined or merged in order to drive the behaviour of the application. By combined, we mean the information must be used together to calculate an output relevant to the application. Analysing two of the example applications, the automatic skid control application needs to combine the data from six separate sensors in order to decide what action, if any, to take to regain control of the automobile. The squash application needs to combine sensor data from four separate sensors to advise the athlete on how to improve their swing. All of the examples given in section 2.1 have the characteristic that the behaviour of the application relies on the output from multiple sensors and that the application must combine this sensor data.

2.2.3 Asynchronicity

All of the applications in section 2.1 interact with the real world and monitor it for events of interest to the application. The method that they use to monitor the environment is through the sensors that the application uses and the application generally does some analysis based on the sensor data from one or more sensor and deduces that an event of interest has occurred in the environment being sensed. For example, in the fall detection application the main event that the application is used to detect is the violent fall of the person wearing the sensor. In the user interface application the movements of the user are monitored and events being detected correspond to particular actions of the user, e.g., panning movements and automatically switching between user modes.

2.2.4 Access to Historical Data

Each of the applications described in section 2.1 must not just detect some event in the environment, they must also then analyse the historical data before the event and use this data to drive the behaviour of the application. This analysis uses the historical data, possibly in conjunction with the present data, to determine the correct course of action. In the car skid example, the application must first detect that a skid is occurring and once this event has been observed must use the historical data from sensors, such as the accelerometer, to determine what corrective action must be taken in order to regain control of the vehicle. In the fall detection example, the system must first detect that a possible fall event has occurred and must then analyse the sensor data from the period leading up to the possible fall event to detect if it was a violent fall and determine if the patient might be in danger.

2.3 Programming Abstraction Support

Abstractions are a vital part of software development. For example, the abstraction of a function in the C programming language, while being basic, is very useful. The developer can divide the application into modules and does not need to worry about the mechanics of how the function is implemented e.g., the state of the calling program or variable management on the stack. This allows the developer to use the abstraction of a function to write better more manageable code, which can be re-used easier. This process of abstraction is evident in many branches of computer science. Programming languages have numerous abstractions, from *types* and *functions* in languages such as C [118] to *objects* and *interfaces* in modern Object-Orientated programming languages such as Java [10]. Abstraction is also prevalent in the operating system community, e.g., inter-process communication in the form of *semaphores* and *pipes* [130], and in the distributed systems community, e.g., *Remote Procedure Call* [102] and *Distributed Shared Memory* [25]. Abstraction is also an important aspect of middleware. Middlewares provide useful abstractions that reduce the burden on the programmer by providing a range of functional and/or non-functional features. These abstractions form the core of the middleware and are the manner in which we access its features. For example, in a broker-based event middleware the abstractions of an event and a broker are the core concepts that programmers use when accessing the services of the middleware and good abstractions are among the main reasons for using the middleware. By using these abstractions, the developer benefits from all the non-functional and functional features associated with the middleware.

Developing sensor-driven applications present additional challenges when compared to developing applications that are not sensor-driven. The manner in which we aim to address these challenges is by proposing a set of additional abstractions that extend event-based programming and which can simplify the task.

In the following sections we propose a number of requirements that programming abstractions to support sensor-driven applications should posses in order to be suitable abstractions for the domain. These requirements are synthesised from the characteristics of sensor-driven applications determined in

section 2.2. The requirements are numbered and divided into two categories: functional requirements (denoted by an F prefix) and general middleware requirements (denoted by an M prefix).

2.3.1 Functional Requirements

In section 2.2 we discussed the common characteristics of sensor-driven applications. In this section we will define requirements for abstractions to support sensor-driven applications that have these characteristics. Any set of abstractions aimed at supporting sensor-driven applications should satisfy all of these requirements.

F1: Combination of Sensor Data

The presence of multiple sensors, and the need to combine information from these sensors, were noted in section 2.2 as two characteristics of sensor-driven applications. Therefore the ability to combine sensor data from multiple sensors is a requirement of middleware to support sensor-driven applications.

F2: Continuous Sensor Data

In section 2.2 we noted that sensor-driven applications need to analyse historical sensor data at points of interest to the application. As the sensors used in sensor-driven applications can be continuous and potentially provide a never ending stream of sensor readings the processing of this historical data is not trivial. A requirement therefore of middleware to support sensor-driven applications is abstractions to support the analysis of continuous sensor data.

F3: Analysis of continuous sensor data from multiple sensors

From section 2.2 it is clear that sensor-driven applications need to both analyse continuous sensor data and combine readings from multiple sensor streams at the same time. The ability to analyse continuous sensor data from multiple sensors is therefore a requirement of any middleware designed to support sensor-driven applications.

2.3.2 General Middleware Requirements

When proposing an abstraction as a solution to a range of problems, it is important to know how widely applicable it is. When discussing the generality of a particular abstraction for developing sensor-driven applications we are interested in the abstraction's generality in a number of different dimensions.

M1: Event Model Generality

As discussed in section 1.2 the event-based approach is the *de-facto* standard approach to building ubiquitous computing applications. Sensor-driven applications are a sub-set of ubiquitous computing

applications so any abstraction proposed as a general abstraction for developing sensor-driven applications should be compatible with the event-based approach. It should also be compatible with the full range of different event systems and event models used in ubiquitous computing applications.

M2: Language Generality

All programming abstractions must map to a specific language in order to be implementable. However it is important that the abstractions are mappable to a large number of different languages in order to be generally useful. Generally speaking, it is desirable that the abstractions should be implementable in as wide a range of programming languages as possible. This allows the abstractions to be used in as many different applications as possible. If the abstraction is not mappable to a wide range of languages, the choice of programming language restricts what abstractions can be used and *vice-versa.* Abstractions intended to be generally applicable in ubiquitous computing applications should map to all languages used to develop ubiquitous computing applications.

M3: Generality across Application Domains

In section 1.1 we noted that there are a large range of ubiquitous computing applications and in section 2.1 we presented four example sensor-driven applications from four different application domains. Any abstraction proposed as a general abstraction for use in ubiquitous computing applications must be general across a range of application domains. When proposing a set of abstractions as a general solution for use in the development of sensor-driven applications the abstractions must be applicable in the majority of application domains.

M4: Platform Generality

Sensor-driven applications are implemented on a range of different platforms. This can be seen in the example applications in section 2.1. The fall-detection system could be implemented on a mote [133] platform. The motes would provide good support for communication and power efficiency while remaining small enough to be worn on the body of the patient for a long period of time. The squash application could be implemented on the Gumstix [70] platform. This would provide enough processing power for the analysis of the squash swing, while also remaining small and light enough to be embedded in the squash racket. The mobile game application could be implemented on the HTC G1 [38] smartphone platform. The HTC G1 comes has three accelerometers, 3 magnetometers and a GPS sensor and has the processing capabilities to support a game. The skid-detection and recovery application could be implemented on an embedded x86 personal computer. The automobile is large enough to easily carry an embedded personal computer and it could also easily supply enough electrical power.

Figure 2.1 shows the sensor-driven application platform spectrum running from embedded systems, through motes and smartphones to applications running on servers. A sensor-driven application can be running on multiple motes, embedded systems or clusters of servers, e.g., similar to the hardware

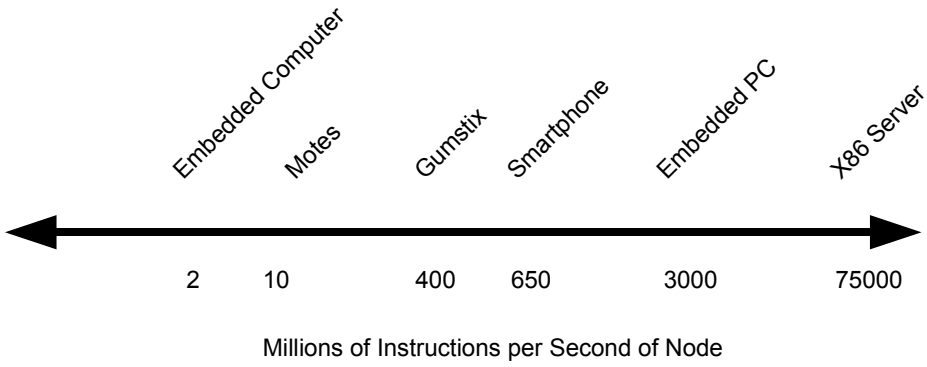

Figure 2.1: The sensor-driven application platform spectrum

configuration of the HiFi system [37].

Abstractions designed to be useful in the majority of ubiquitous computing systems must be suitable for use on the range of platforms on which ubiquitous computing applications are run. Some abstractions, while being useful and general, incur a significant run time cost and therefore are not suitable for certain platforms. As we can see from figure 2.1 a large amount of sensor-driven applications are implemented on resource-constrained platforms, e.g., mobile phones and embedded computers. Therefore any abstraction aimed at being a general abstraction for use in ubiquitous computing applications must be suitable for use on resource-constrained platforms.

2.4 Conclusion

In this chapter we have presented four examples of sensor-driven ubiquitous computing applications. They use a wide array of different sensors and software techniques to solve complex problems. All of the applications are sensor-driven applications in the sense that they combine multiple sensors to drive the behaviour of the application. Each of the applications is in a different application domain. These applications are used to motivate our description and analysis of sensor-driven applications. We have analysed these applications and derived from them common characteristics of sensor-driven applications. These characteristics were used to synthesise the key requirements for middleware intended to support sensor-driven ubiquitous computing applications. These requirements have been divided into two categories: Functional Requirements and Middleware Requirements. We have summarised

	Functional Requirements
F1:	Combination of Sensor Data
F2:	Continuous Sensor Data
F3:	Analysis of Continuous Sensor Data from Multiple Sensors
	General Middleware Requirements
M1:	Event Model Generality
M2:	Programming Language Generality
M3:	Application Generality
M4:	Platform Generality

Table 2.1: Requirements of Middleware to Support Sensor-Driven Applications

the requirements in table 2.1. The requirements sub-divide into two discrete groupings functional requirements and general middleware requirements. The requirements for middleware to support sensor-driven applications will be used throughout Chapter 3 in order to analyse the state of the art for sensor-rich applications. They will also be used in the analysis and evaluation of our contribution in Chapters 5 and 6.

Chapter 3

State of the Art

Chapter 1 introduced event-based programming and stream processing as two techniques for building ubiquitous computing applications and stream-based applications respectively. As stated in section 1.4 we aim to combine the two approaches by taking suitable abstractions from the domain of stream processing and using them to extend event-based programming. Chapter 2 discussed sensor-driven applications and derived a set of requirements for abstractions aimed at supporting them. In this chapter we review the state of the art in abstractions used to develop sensor-rich applications. By reviewing existing abstractions used to develop sensor-rich applications we aim to analyse these abstractions and using the requirements derived in section 2.3 determine how well they support the development of sensor-driven ubiquitous computing applications.

3.1 Review Methodology and Work Selection Criteria

In section 2.3 we discussed characteristics of programming abstractions which are desirable for supporting sensor-driven ubiquitous computing applications and derived a set of requirements for such abstractions. We use these requirements to analyse the state of the art for abstractions used to develop sensor-rich applications. Although the work in question may not have been designed with these features in mind it is still a worthwhile exercise to review the projects in this manner to understand how useful the abstractions provided are for implementing sensor-driven applications.

There is a very large body of work that could be selected for this state of the art review. Because the nature of our work involves extending event-based programming with work from other fields, there are a number of different fields that must be covered. Section 3.2 reviews the traditional event-based approach to developing applications. Event systems which are designed to support event composition are also important to the work in this thesis and this work is reviewed in section 3.3. Complex event processing is a body of work that merges multiple events which may or may not originate from sensors and is reviewed in section 3.4. Data stream processing is another major research field related to processing streams of data including sensor streams and this research field is reviewed in section 3.5. There are a number of other pieces of work which aim to merge event-based systems and stream

processing and we review this work in section 3.6. Finally we discuss the sensor database abstraction, an alternative approach to combining sensor readings which is popular in wireless sensor networks, in section 3.7.

There are clearly a large quantity of projects that can be reviewed with this format, far too many to fit in this thesis. Therefore to reduce the amount of projects to be reviewed we review the two most influential and relevant projects from each field.

3.2 Simple Event-Based Systems

Event-based programming is a very large and mature research field in computer science, and there are many significant pieces of work [112, 15, 92, 104, 129, 99, 101, 40, 19, 119, 52, 57]. As stated in section 1.2 event-based programming is a natural fit for dealing with sensor data as each datum can be modelled and communicated as an event. Event-based programming is also a very common programming paradigm in the field of ubiquitous computing. In addition to the basic event-based functionality most event-based systems provide additional functionality to improve their performance in a particular scenario or to make them more suitable for a particular domain. When analysing simple event-based systems we limit our discussion to event-based systems that have no explicit mechanism for creating complex events, i.e., events which are formed from the combination of one or more events.

3.2.1 JavaBeans Event Model

Java is a popular object-oriented programming language developed by Sun Microsystems. JavaBeans is a component model for Java and defines its own event model [103]. The event model is designed for small centralised systems such as window toolkits but can be used in a distributed fashion by utilising the Java Remote Method Invocation (RMI) system.

Events are one of the core features of the JavaBeans architecture. They are one of the mechanisms used to allow components of the architecture communicate with each other. JavaBeans are used in a wide variety of applications from applets to desktop applications. Figure 3.1 shows an overview of the JavaBeans event model. In the event model, event notifications are communicated from event sources to event listeners by invocation of methods on the listener objects. Listeners wishing to register their interest in events must implement the EventListener interface. State associated with an event notification is encapsulated in an event object that inherits from java.util.EventObject and is passed as the sole argument to the event method. Event sources must provide methods by which event listeners can register and de-register themselves as listeners for particular events.

The JavaBeans event model does not explicitly support filters, however it does support a type of component called Adaptors. Adaptors are objects that can be placed between event sources and listeners. They can implement additional functionality in the event system, such as filtering and event queueing.

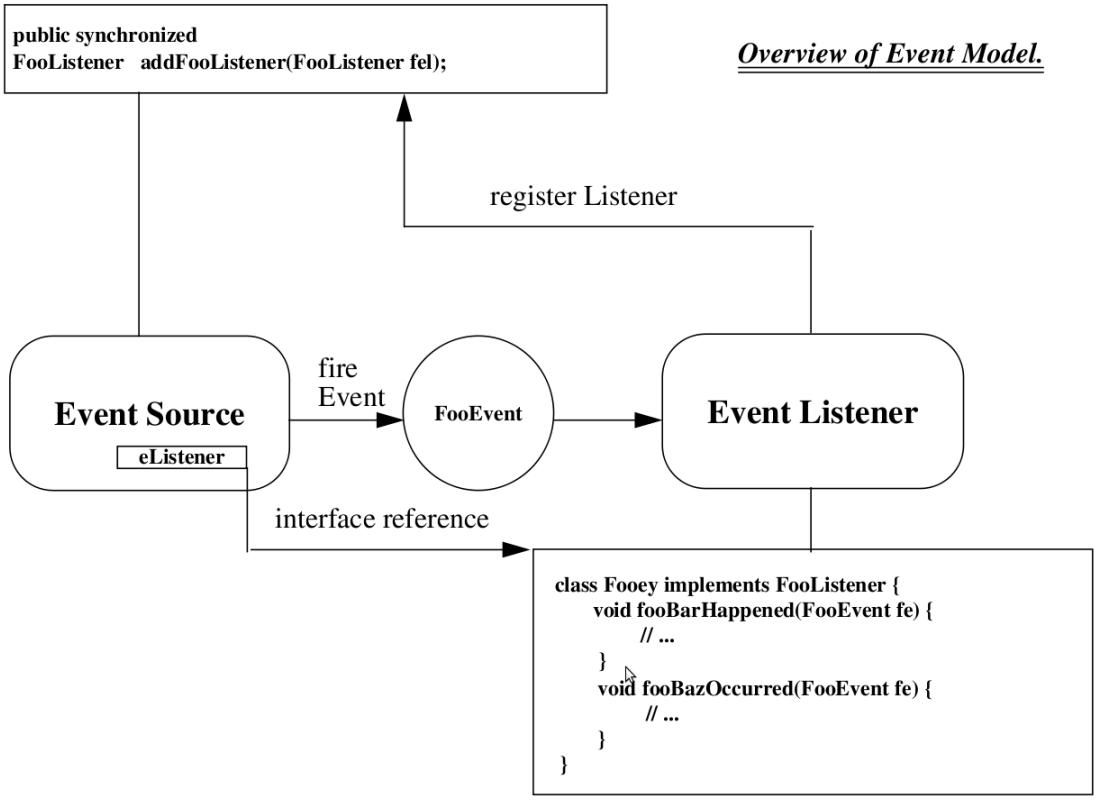

Figure 3.1: The Java Beans Event Model [103]

Discussion

The JavaBeans event model is a simple event model designed for small-scale applications that are not distributed. The synchronous nature of the event delivery mechanism coupled with the fact that event sources must call the handler function of all the event listeners in sequence means that even if RMI is used event sources must wait until the event is delivered before they can proceed with their own execution and notifying the remaining listeners which incurs significant performance penalties for event sources with multiple consumers. Anonymity between event source and listener is not provided by the model. Even when adaptors are used a reference to the adaptor is used as the destination of the event, and the event listener must still know the identity of the event source and register directly with it.

The JavaBeans event model provides no means to combine multiple sensor streams. Any combination of multiple events would need to be done in an ad-hoc manner perhaps by a number of event listeners updating a shared data structure. There is no in built mechanism in the JavaBeans event model for dealing with historical events or continuous event streams.

3.2.2 STEAM

Scalable Timed Events and Mobility (STEAM) [99, 100, ?] is an event based middleware from the Distributed Systems Group in Trinity College Dublin. It is designed for a range of applications

including indoor and outdoor smart environments, augmented reality systems and traffic management scenarios [100]. STEAM is designed with the notion that mobile devices are generally more interested in events which occur in closer proximity to the producer and that certain events can be filtered in this manner. It extends the standard event model with the notion of proximity filtering which restricts event delivery within a geographical space. STEAM is designed for use in mobile ad-hoc networks although it can be used with standard infrastructure-based networks also.

Event Model

STEAM uses an implicit event model [101], meaning that event consumers subscribe to event subjects and not with a mediator or with another node. This is in contrast to peer-to-peer and mediator-based models. The implicit event model is preferred in this instance because in a mobile ad-hoc network one cannot always assume the presence of delivery services in the network (e.g., a message broker) and so each node must be able to receive and deliver its own messages.

Events in STEAM contain a subject and a content. The subject denotes the name of the event that has occurred, e.g., a door opening event, and the content contains a list of attributes that refer to properties of the particular instance of the event, e.g., the location of the door and the time of the event. STEAM producers and consumers must agree *a priori* on definitions of events and schemas for event attributes. Sensor data can easily be encapsulated in a STEAM event where the event subject relates to what sensor produced the data and the content carries the sensor data.

Events in STEAM are not timestamped and the STEAM event model does not explicitly deal with the timing of events. Out of order events are handled when they arrive at the consumer and no attempt is made to re-transmit events that have not been delivered to a consumer.

Discussion

The STEAM event middleware provides an event system to facilitate event-based programming. It is specifically designed for the mobile ad-hoc domain. A significant subset of ubiquitous applications have mobility as a main concern and the ad-hoc communication paradigm is suited to addressing these issues. Therefore STEAM would seem to be suitable to address this subset of ubiquitous computing applications. STEAM has been implemented on a range of platforms including severely resource constrained platforms.

STEAM has been implemented in a range of languages including Java and C. While it is designed for mobile ad-hoc networks STEAM can be used on standard networks so is capable of supporting a wide range of event-driven applications. STEAM does not provide mechanisms for combining events, nor does it provide abstractions for dealing with historical sensor data. STEAM is suitable for implementation on the complete range of platforms in the sensor-driven application platform spectrum discussed in section 2.3.2.

3.3 Event Systems with support for Composite Events

Composite events are events that are composed from two or more other events. There are many event systems which support composite events [123, 86, 114, 97, 16, 89, 124]. We have selected the Cambridge Event Architecture [16] and Composite Event Detection as a Generic Middleware Extension [114] as the two projects to review from this body of work. We have selected these projects because they have a representative set of functionality and have similar goals to this thesis in that they intend to create a generic middleware extension for use with a wide range of host middlewares.

3.3.1 Cambridge Event Architecture

Bacon et al. [16] propose an event architecture called the Cambridge Event Architecture (CEA) which is intended as an extension to any synchronous object-oriented middleware to support asynchronous operation. It is intended for a wide range of application domains including healthcare, location-aware systems and network-monitoring systems. CEA supports asynchronous operation of applications by using events, event classes and event occurrences as object instances [16]. CEA uses a publish-register-notify paradigm with event object classes and also supports source-side filtering based on parameterisable templates. Event objects publish their interfaces specified in Interface Definition Language (IDL) including the events they will notify. CEA also supports access control through the use of the OASIS service [58].

Event Mediators

CEA contains the abstraction of event mediators. Event mediators are intermediaries between producers of events and their consumers. The event source notifies the mediator which then notifies the consumers. They can be used to implement a range of functionality, e.g., providing a higher level interface to event consumers than might be provided by a more primitive event source. They can also be used to add event filtering capabilities or to support disconnected operation of mobile devices.

Event Composition

The CEA also supports a composite event service and provides composition operators and a language for specifying composite events. Figure 3.2 shows the composite event service in CEA. CEA also uses stream semantics to model incoming events. Five operators are defined for specifying composite events: WITHOUT, SEQUENCE, OR, AND and FIRST. Unfortunately very little detail about the composite event system is available, e.g., relating to time model or composition operators.

Historical Events

CEA supports an event persistence mechanism which is a service provided by the event store manager. The event store manager subscribes to events of interest and stores these events for later retrieval. The event store can subsequently be queried using a mechanism similar in concept to the process of defining composite events where an event filter is applied to a collection of events.

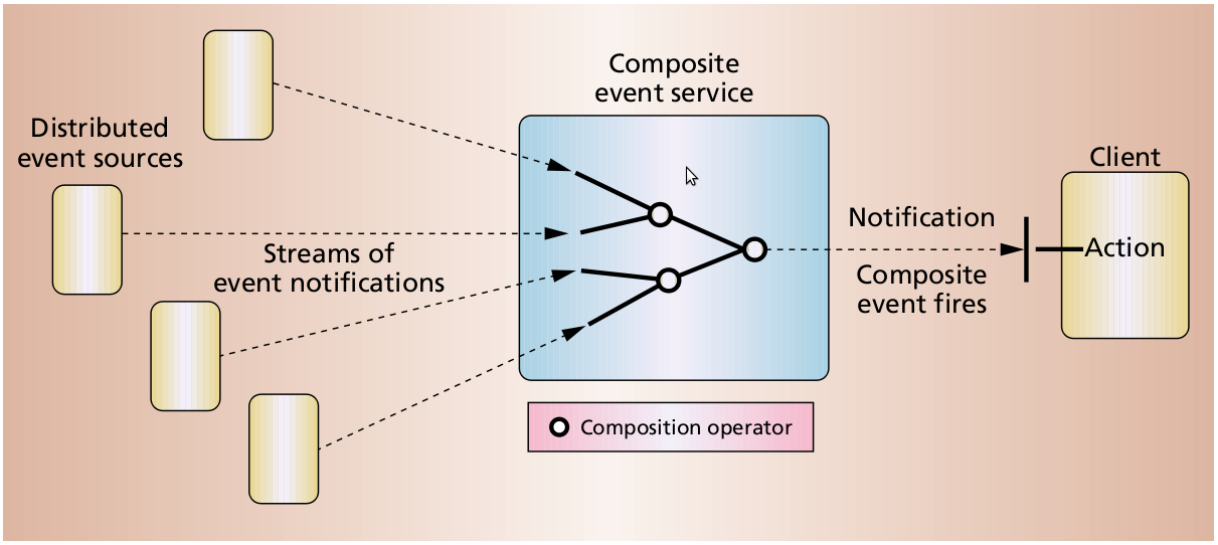

Figure 3.2: Composite Event Detection in CEA [16]

Discussion

CEA is an effective way of adding asynchronous communication capabilities to an otherwise synchronous system. Event mediators allow additional functionality to be incorporated in the event services and it is easy to imagine a range of additional services being implemented in this manner including event filtering. The system also supports composition of events which should allow it to be used to develop more sophisticated event-based applications.

Very little information on the composition of events is provided. The time model for the event system is not available and this would be useful in conjunction with knowledge of the event composition language in determining how expressive the event composition system is. The composition operators are limited and no scope is available for implementing additional operators or for custom operators.

Historical data is accessed using a traditional database-style approach. This approach is only suitable for an event system with a very low event rate. A moderate rate of events would quickly exhaust the database storage space and the process of storing events and then querying the system would significantly increase the processing time for applications which relied on this data. This method of historical data access would be insufficient for implementing a stream processing style application in which a stream of events needed to be monitored and analysed.

With respect to sensor-driven applications, the event composition ability of the architecture could support limited sensor-driven applications, however more sophisticated sensor-driven applications might prove difficult to implement because of the lack of expressiveness in the event composition language. There is limited support for accessing historical events and as noted earlier any more than a moderate event stream would create difficulties for the application. It does not appear to be possible to perform custom analysis outside of the limited set of operators provided by the event composition language. The inclusion of event composition features and access to historical data, however limited, shows that the authors consider this functionality to be a useful addition to event-based systems.

It is unclear as to the resource requirements of the CEA middleware, however the event store

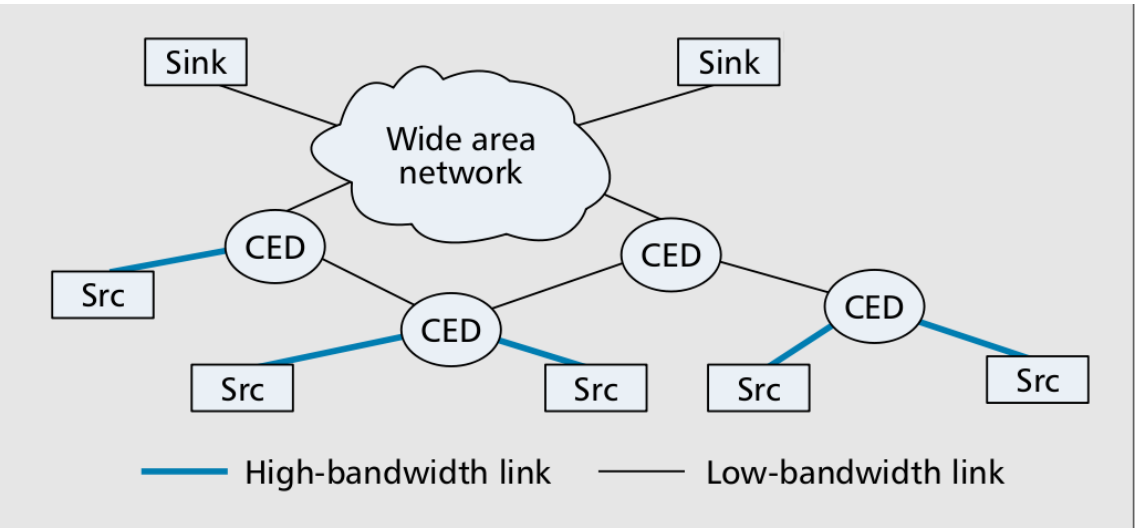

Figure 3.3: Illustration of distributed composite event detection [114]

capabilities would imply that sufficient storage space is available to store a reasonable number of events. The object oriented nature of the system does not lend itself it to implementation on severely resource-constrained platforms at the lower end of the sensor-driven application spectrum.

3.3.2 Composite Event Detection as a Generic Middleware Extension

Pietzuch et al. [114, 113] propose a generic extension that supports composite event detection. The extension relies on a minimum set of requirements of a distributed event system and provides support for composite event detection. The work is motivated by the fact that in a large scale distributed event-based system, event sinks performing composite event detection can quickly become overwhelmed by primitive events and would benefit by receiving higher level events. Distributing composite event detectors in the network near the source of the events would result in a reduction of events being propagated to the event and would reduce bandwidth congestion in the network around the event sink node. This is shown in figure 3.3.

Requirements

The extension relies on a minimum set of requirements from the event system to be extended. The underlying publish/subscribe system must be able to allow subscriptions to events, publish notifications of events and relay events from event source to the event consumer. The relationship with the host event system is shown in figure 3.4. The event system must also support a particular time model and event model. In the extension each event has an associated timestamp. There is a partial order on all timestamps in the system. This is extended using a tie-breaker convention so that there is a total ordering on all events in the system. The prerequisites that the event model relies on are that each event has a timestamp and that each event can be consistently ordered for each subscriber (e.g., by timestamp, source IP address and local event generation count).

Any event system that satisfies these criteria is suitable for extension and the authors have extended

the Java Messaging Service (JMS) [131] and the Hermes event system [112].

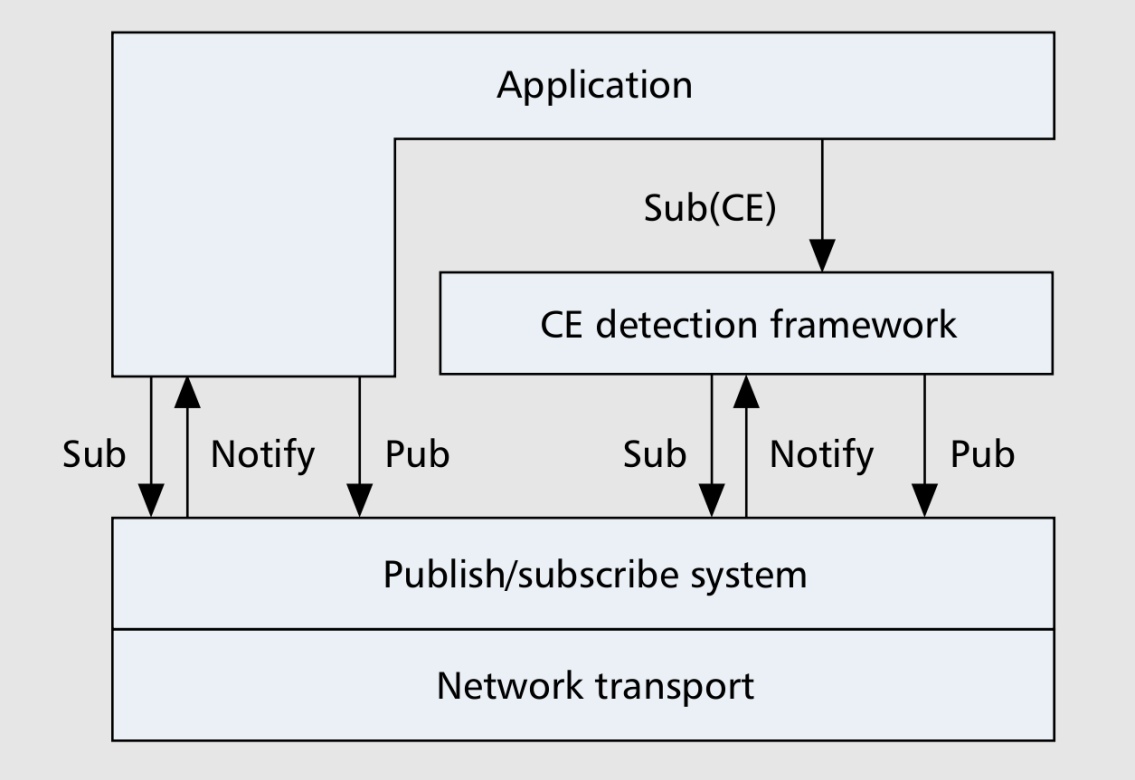

Figure 3.4: Interface with the host event-model [114]

Composite Event Language

The extension provides a custom language for specifying event patterns. The evaluation of event patterns uses finite state automata extended to deal with the temporal aspect of the events. The core language supports expressions similar to regular expressions and support operators such as sequence and iteration. The language also specifies a timing operator which matches if the corresponding event combination occurs within the timing interval. The language is designed to be decomposable into sub-expressions to facilitate distribution throughout the network.

Discussion

Pietzuch et al. propose an interesting extension to event-based systems. Composite event detection is a feature that is in demand in event systems and there have been several notable composite event systems. However this work allows existing event systems to be extended which is important for a number of reasons. Existing event systems that may already be developed and well understood can be re-used thus saving a large development and retraining effort. Many systems desire composite events but also need characteristics which are already implemented and well understood, e.g., distribution. This extension allows these systems to be used and extended instead of attempting to add these features to composite event systems which may prove impossible.

This system is reasonably general and can be used with a large amount of existing event models. While it makes certain demands of the underlying event model and time model, these demands are not overly restrictive. The extension does specify a custom composite event detection language for specifying composite events. This custom language is not very accessible or intuitive to end users or developers and instead is designed to be efficiently implemented. The authors have recognised this shortcoming and have specified providing support in other languages as future work [114].

This system is designed with large scale systems in mind, however there appears to be no reason why it could not provide composite event detection support for a large range of ubiquitous computing applications, when coupled with the appropriate event system.

The work supports composite event detection and with appropriately modelled sensor data could be used to combine multiple sensor streams, and indeed this is specified in a scenario by the authors [113]. The system provides limited support for analysing historical sensor data when detecting composite events and provides no ability to perform analysis on the sensor stream once a composite event is detected.

The extension has not been implemented on resource-constrained systems, however it appears likely that the extension would be suitable for all but the lower end of the spectrum of resource-constrained platforms used in ubiquitous computing systems.

3.4 Complex Event Processing Systems

The event systems and architectures we have reviewed thus far have been mainly concerned with the creation of event-based applications and the programming of systems using them. These systems have been implemented in traditional programming languages and typically take the approach of allowing events to be used as a programming abstraction. Event processing systems are a class of event systems designed to analyse large quantities of existing events. They are analogous to databases in traditional applications. They generally take the approach that there is a large volume of events which must be processed to extract meaningful higher level events. They provide a system which can be programmed using a custom declarative language to perform analysis on a collection of events. Complex event processing systems support a large class of applications usually ones in the business application space, e.g., warehouse stock monitoring and automated stock trading.

There are a large number of event processing systems available both in research and in industry. We have chosen two systems SASE and Cayuga as being representative of the feature set available and analyse them in this review.

3.4.1 SASE

SASE [54] is a complex event processing system which operates on event streams. It provides an event processing language similar in syntax to SQL and other Stream processing and event processing languages. This language SASE+ supports kleene closure over event streams. The system is designed for applications such as financial services, RFID-based inventory management, click stream analysis,

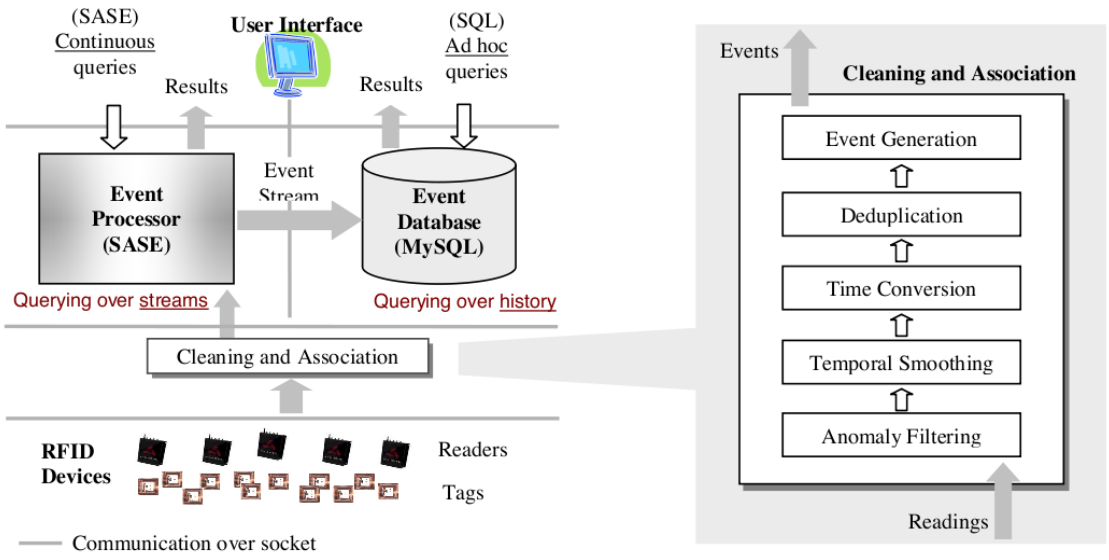

Figure 3.5: The SASE architecture [54]

electronic health record systems, etc. [137]. The system architecture is shown in figure 3.5. The system is designed for high-performance event stream processing. Two specific challenges it addresses are high volume streams and efficient extraction of events from large sliding windows.

Data Model

In SASE each event in an input event stream represents an atomic occurrence of interest at an instant in time. Events have a type which describes the set of attributes that a particular event must contain. Each event in the system has a timestamp and timestamps are considered to be monotonically increasing, which gives rise to a total order of events. Input events in SASE are simple events while output events can be more complicated, however the eventual aim of the authors is to enable input event streams to support the same range of events as output event streams which would allow queries to be composable.

Queries

Queries in SASE+ take the form FROM-PATTERN-WHERE-WITHIN-HAVING-RETURN. The FROM clause specifies the input stream which defaults to the standard input for the system. The PATTERN clause declares the structure of the pattern and any variables that will be needed to refer to the individual events. The WHERE clause, if present, defines the events that are relevant to the pattern using value-based predicates e.g., WHERE stock.price $>$ 10. WITHIN defines a time based window over which the pattern matching must take place and RETURN transforms each pattern match into an event to be returned.

Sliding Windows

SASE provides time based sliding windows over event streams. These sliding windows allow event pattern recognition to occur over a fixed time period. This could be used in sensor-driven applications to detect event patterns in a relevant time period over a sensor stream.

Discussion

SASE provides powerful event pattern detection. The sliding window support allows the event pattern to be specified over a time period and can be used to detect complex events. Tuple-based sliding windows, i.e.,windows who's membership is defined by being a fixed number of tuples, are not supported, however, and the event patterns are limited to being matched across one time-based sliding window. This limits the flexibility of the patterns that can be detected, e.g. we cannot store the previous 200 events from a high frequency event source with 5 events from a less frequent source, we can only store the events that have happened in a fixed time period.

SASE does not provide the extensibility of user-defined functions. This means that system is limited to detecting event patterns and cannot perform any processing on the event streams once it has detected that a complex event has occurred. This severely limits the usefulness of the system for implementing sensor-driven applications as performing custom analysis of event data across event streams is impossible with the system.

In terms of resource usage SASE lies between the more resource-hungry stream processing systems and the lower level event systems. It has not been designed for or implemented on resource-constrained platforms and it is difficult to see how the SASE+ language and the SASE system could be implemented effectively on such a platform.

3.4.2 Cayuga

Cayuga [20, 42, 41] is a general purpose event monitoring system from Cornell University. It is designed to support a large class of applications including supply chain monitoring of RFID tagged products, real-time stock trading, monitoring of large computer systems to detect malfunctioning or attacks, and monitoring of sensor networks [20]. The CAYUGA system architecture is shown in figure 3.6. Event receivers (ERs) receive the events from the event stream, deseriealize them, and place them in the Priority Queue (PQ) and the Heap. The query engine processes the events and can resubscribe its results on the PQ or submit them to the client notifier threads (CNs). The Cayuga system monitors event streams and detects event patterns over these streams. It uses the presence or absence of these event patterns to infer some additional knowledge about the system and can publish this as an event. Cayuga is designed to scale in both numbers of subscriptions and the arrival rate of events. The authors consider the work to be positioned between traditional publish/subscribe systems and stream databases systems, offering more expressiveness but less scalability than traditional publish/subscribe and less expressiveness but better scalability than stream databases [41].

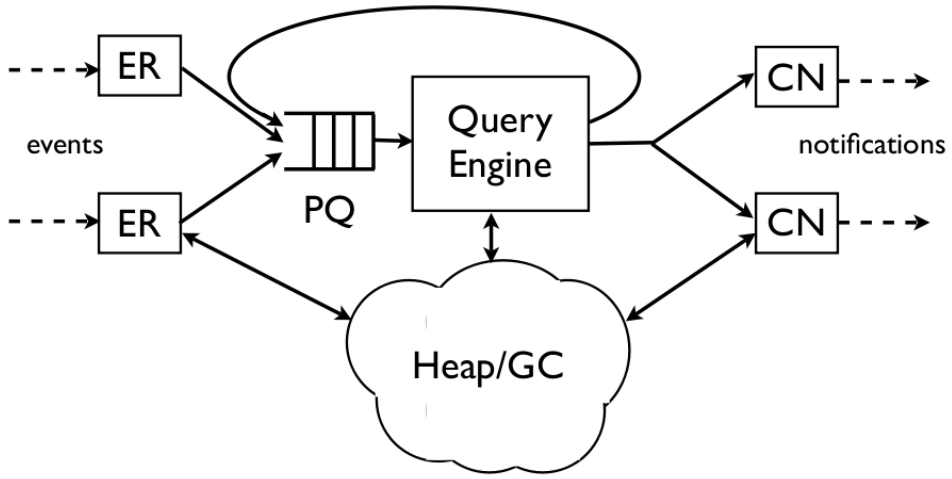

Figure 3.6: The CAYUGA system architecture [20]

Event and Time Model

In Cayuga an event stream is a possibly infinite set of event tuples. These event tuples have a set of attributes, a start time of the event and an end time of the event. It is assumed that each event stream has a fixed schema and that events arrive in temporal order. Event streams can contain events with non-zero duration, overlapping events and simultaneous events.

Operators

The Cayuga Event Language (CEL) is based on the Cayuga algebra. CEL is a mapping of this query algebra into a SQL-like syntax. Each query in CEL is in the form of SELECT - FROM - PUBLISH. The SELECT clause in CEL is similar to the SELECT clause from SQL and is used to specify the attribute names in the output schema as well as aggregate computation. The PUBLISH clause allows the output stream to be named. The FROM expression forms the core of the query and is formed from a combination of one or more from three operators; FILTER, NEXT and FOLD. Each of these operators produce an output stream from one or two input streams. The FILTER operator selects those tuples from its input stream that satisfies the supplied predicate and allows these tuples to proceed. The NEXT operator takes two input streams. It evaluates a predicate based on the first stream, and if it is true combines this tuple with the next tuple from the second stream that satisfies a second predicate. The FOLD operator is the most complicated of the three, it acts like an iterative version of NEXT and takes three predicates. The first predicate specifies the condition for choosing input events in the next iteration, the second predicate specifies the stopping condition for the iteration and the third predicate specifies the aggregate computation between iteration steps.

The Cayuga system allows user-defined functions. These functions are not well documented in the literature, however from examining the source code for the project they appear to be only definable on two arguments and have no access to historical stream data.

The Cayuga system allows resubscription of events allowing queries to view the output of other queries as their inputs. This allows for complex queries to be built from simple ones and greatly increases the expressiveness of the system.

Discussion

The Cayuga system uses events which is a highly popular mechanism for developing ubiquitous computing applications. The events in the system are also useful for modelling sensor data. Cayuga can perform state-full queries over event streams. Cayuga uses a custom query language which is limited in its expressiveness. Cayuga scales very well to a large volume of concurrent queries and subscriptions, however the trade off that is made to support this scalability is limited expressiveness of the language and a restriction in what can be performed with the system. In terms of application support the operators in CEL limit the applications to simple event-monitoring applications, where a pattern of events must be detected. In terms of sensor-driven applications this falls short of what is required.

Cayuga provides support in CEL for combining multiple event streams and therefore is useful for combining sensor data. In terms of historical sensor data Cayuga is more limited. The *Dur* construct can be used to scope a query for a particular length of time and implements a time-based sliding window. However as with the SASE system in section 3.4.1 limiting the system to time-based sliding windows severely restricts the usefulness of the system as does the fact that only one window is permitted for the entire query, i.e.,you can not specify individual windows for each stream.

Cayuga while being able to detect event patterns in event streams does not support analysis over these event streams once the event has been detected. This limits the system to once which simply detects composite events.

The CAYUGA system has not been designed with resource-constrained platforms in mind, and it is difficult to see how the system could be implemented on such platforms.

3.5 Stream Processing

Data stream processing is a body of work that has developed from the active database community. Traditional databases are unsuitable for dealing with continuous streams of data. The main reasons for this is that traditional databases must first store the data in the database and then query it at some later time. This process is not suitable for an application with a large amount of streaming data as the database quickly runs out of storage space. A large class of applications need to query streaming data in almost real-time and be updated with results of these queries as soon as they are available. Applications such as large-scale military applications, real-time stock control systems, share monitoring applications, RSS feed monitoring services all require near real-time results from the combination of multiple streams of incoming data. This is in contrast to the approach of storing this streaming data in a traditional database and running nightly or weekly queries on the dataset to extract pertinent information. The data stream management approach is to register continuous queries

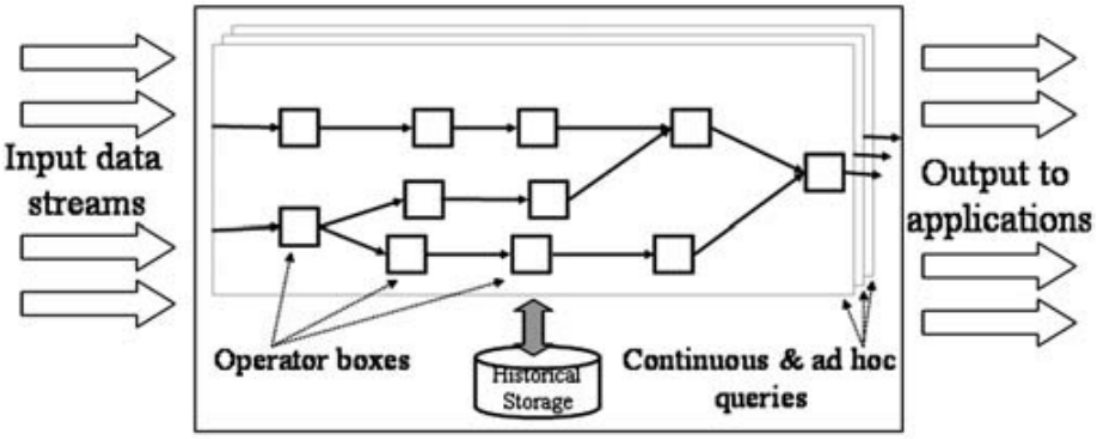

Figure 3.7: Aurora system model [1]

with the Data Stream Management System (DSMS). These continuous queries are then evaluated over some window of incoming streaming data and possibly traditional relational tables and the results are then presented to the user. A large variety of additional features are also available to the user such as query optimisation, quality of service guarantees and partial results of queries. There are a large number of stream processing projects [1, 82, 12, 13, 93, 31, 32, 33, 85]. We have chosen Aurora and STREAMS as the two projects to review from this body of work as they are highly influential and implement a representative range of the abstractions from the domain.

3.5.1 Aurora

Aurora [1, 2, 120, 23] is a data stream processing system aimed at supporting real-time applications on data streams, from sensors or other sources. It is specifically aimed at applications which monitor continuous streams of data. The Aurora system model is shown in figure 3.7. Aurora supports continuous queries specified by a system administrator using a graphical user interface. The authors of Aurora have identified five features that distinguish it from other DSMS, they are workflow orientation, a novel collection of operators, a focus on efficient scheduling, a focus on maximising quality of service, and a novel optimization structure [120]. Aurora was extended by combining it with the Medusa [120] system to create Borealis [6, 17] which is a distributed stream processing system. However the core abstractions were not significantly changed with this extension so here we review the original Aurora system.

Aurora is designed as a workflow system. The application administrator uses a GUI to design an Aurora system using "boxes" and "arrows" which correspond to streams and operators. Aurora has sophisticated techniques for optimisation, scheduling and providing quality of service guarantees to its clients. However, discussion of these features is not relevant to the topic at hand and therefore excluded from this review.

Stream Model and Operators

Streams in Aurora are modelled as an append-only sequence of tuples with uniform type (schema). Each tuple has application specific data fields. Each tuple also has a timestamp which specifies its time of origin in the Aurora network.

Aurora supports a set of 8 operators; Filter, Map, Union, BSort, Aggregate, Join and Resample. These operators take one or more streams of data and output a new stream of data, potentially preserving features of the incoming stream such as timestamp and other attributes or potentially creating entirely new output tuples.

The Filter operator acts in a similar manner to a case statement in a procedural language such as C. A number of predicates and output streams are specified in the operator and when an input tuple arrives in the operator it is outputted on the first stream whose predicate it matches in the order in which it is specified. Output tuples have the same schema and values as input tuples including timestamp and QoS.

Map is a generalized projection operator whose output tuple values are functions over tuples on the input streams. It can have a different schema and values to those of the input stream, but timestamps are preserved in the corresponding output tuples.

Union is used to merge two or more input streams into a single output stream. Union outputs all the input tuples it processes therefore all output tuples will have the same schema and values as the input streams.

The remaining operators are order-sensitive and can only be guaranteed to execute in finite buffer space and finite time if they can assume a particular ordering of their input streams. This is communicated to the operator using order specification arguments.

BSort is an approximate sort operator that operates on a subset of a datastream. It maintains a buffer of the subset and always outputs the smallest value from the buffer on the output stream.

Join is a binary join operator that combines a sub sequence of two streams using a predicate over pairs of tuples from the two input streams.

Resample is an operator which can interpolate between tuples using some user-defined interpolation function.

Aggregate applies "window functions" to sliding windows over its input streams. A "window function" is either a SQL-style aggregate operation e.g., Avg or a Postgres-style user-defined function. The use of custom user-defined functions is claimed to allow the operator set to be Turing complete [1].

Windows

Aurora has support for a number of different windowing mechanisms. It applies a function to a window and then advances the window using a particular policy. Slide advances the window downstream by a certain number of tuples, e.g., it could be used to maintain a window over the previous hours worth of stream data. Tumble is similar to the slide mechanism however unlike Slide no tuples are shared

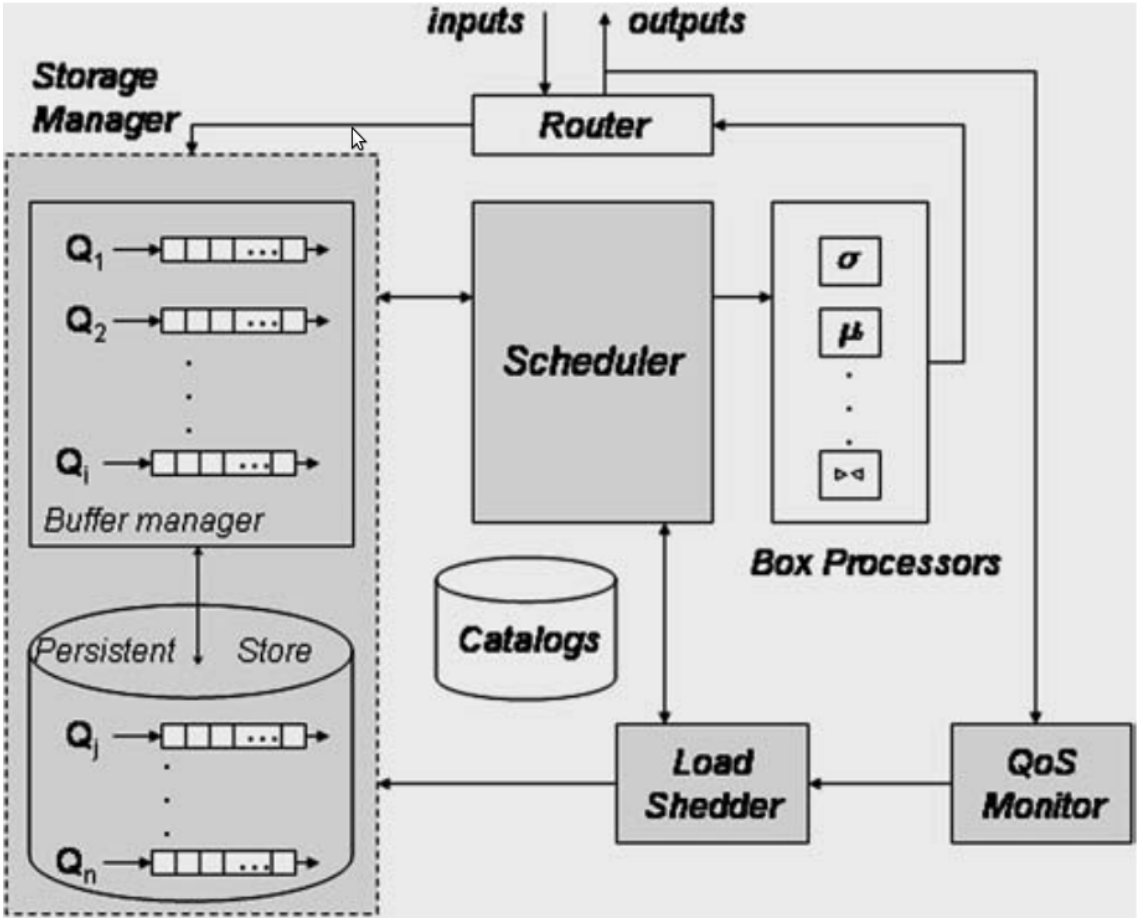

Figure 3.8: The Aurora system architecture [1]

between adjoining windows. This effectively partitions the input stream into disjoint windows. This could be used for example to calculate daily stock indexes or other calculations over fixed windows. Latch resembles Tumble but can also maintain internal state between calculations. This feature can be used for example to maintain a history of maximum or average values. Finally Resample produces a partially synthetic stream by interpolating between the actual input stream readings [23].

Discussion

The Aurora stream processing system is clearly an advanced data stream management systems which provides a large number of features for developers of sensor-based systems. The system deals in tuples and the notion of streams of tuples. The Aurora system is quite general in terms of application and could conceivably support the full spectrum of applications from wireless senor applications to augmented home and pervasive healthcare applications. The Aurora system is designed to combine streams of data and it performs well at this task. It provides several useful operators such as join operators for combining sensor streams and custom operators which can act on stream data. It also has several abstractions for dealing with historical sensor data and in particular its windowing specification ability is very advanced allowing a large range of specialised windows. The functionality of Aurora can be extended by the developer by writing user-defined functions. Through the use of

custom operator extensions Aurora facilitates the analysis of continuous sensor data from multiple sensors, which essentially allow the developer to extend the Aurora system as much as they desire. In general Aurora provides good support for programming with sensor streams. It provides a small set of operators however for applications wishing to do more than basic analysis of stream data custom operators are required. User-defined custom operators greatly increases the expressiveness of the Aurora system however it is restricted to one language and the developer must not just focus on writing code in the language of their application but also on extending the Aurora system. Queries in Aurora may be specified both graphically using the GUI and in XML. Neither of these two options are congruent with the approaches currently used across the field of ubiquitous computing development. Aurora was not designed to run on resource-constrained devices, and it is difficult to conceive of how much of its functionality, such as Scheduler, QoS manager, Overload detection and GUI could be implemented on resource-constrained platforms typical of the mid to lower end of ubiquitous computing systems.

As Aurora is not an event-based system, for the purpose of this thesis it is illustrative to consider how it might map to one. In terms of sensor data in event-based systems the tuples used in Aurora could be mapped to events, with each tuple being encapsulated as the content of an event and the type of the event corresponding to the name of the stream. If we were to consider a mapping from the abstractions provided in Aurora to event-based systems, operators which take one stream would correspond to event handlers, while operators with more than one stream would correspond to a different kind of handler, one which took two or more events as parameters. The windowing facilities could be considered to allow handlers to refer to previous events that had occurred.

The commercial StreamBase project [73] is similar to the Aurora work and has many of the same technical advisors and features.

3.5.2 STREAM

The Stanford Stream Data Management System (STREAM) [8] is a general purpose DSMS that supports a large class of declarative continuous queries over continuous streams and traditional data sets [8, 13]. It aims to support modern applications in domains such as network monitoring, financial analysis, manufacturing and sensor networks [13, 8]. Their design and construction of a new DSMS from scratch was motivated by two fundamental differences they identified between Relational Database Management Systems (RDMS) and DSMSs. A DSMS must handle multiple continuous, high-volume, and possibly time-varying data streams in addition to managing traditional stored relations. Also due to the continuous nature of data streams, a DSMS needs to support long-running continuous queries producing answers in a continuous and timely fashion [53].

Description

To perform continuous queries over streams a custom declarative language called Continuous Query Language (CQL) is provided which is an extension to SQL for supporting streams [9]. The design of

CQL is motivated by leveraging existing well-understood relational semantics and relational rewrites and execution strategies. CQL therefore has a strong relational foundation and takes the approach of providing additional operators which convert streams into relations and convert relations into streams. This allows streams to be mapped to relations and then operated on using well understood relational operators before being optionally converted back into streams. It also allows for easy integration with traditional relational tables.

Stream Model

STREAM combines relations and streams and must have an internal representation of both that allows this to be expressed. Streams are modeled as a bag of tuple-timestamp pairs which is represented as a sequence of timestamped tuple "insertions". A relation is a time-varying bag of tuples which can also be represented as a sequence of timestamped tuples except in this case we have "insertion tuples" and "deletion tuples" which capture how the relation changes over time. STREAM therefore has a common representation of for streams and relations: sequences of tagged tuples. The tuple contains the values of the attributes in the schema and the tag contains the timestamp and whether the tuple is an insertion or a deletion. These tagged-tuple sequences are append-only and are always in nondecreasing order by timestamp [9].

Stream-to-Relation

There is one stream-to-relation operator implemented in STREAM. *seq-window* is a sliding-window operator which creates relations from a portion of the entire stream. It supports tuple-based, time-based and partitioned window specifications [9]. Tuple-based sliding windows provide the last N tuples as a relation, where N is supplied as a parameter to the operator. Time-based windows create a window based on the tuples in the amount of time specified as a parameter to the operator. Partition-based windows take as parameter an integer N and a set of attributes. It is analogous to the Group-By operator in SQL and creates a number of sub streams based on equality of the attribute list. It then creates an N sized tuple based window over these individual streams and performs a union of windows to create the output relation [9].

Relation-to-Stream

STREAM provides three relation-to-stream operators. Istream stands for "insert stream" and is applied to a relation and outputs a tuple whenever a new tuple is inserted into the relation. Dstream stands for "delete stream" and outputs a tuple whenever a tuple is deleted from the relation. Rstream stands for "relation stream" and outputs a tuple for every element in the relation.

Relation-to-Relation

CQL uses SQL constructs to express relation-to-relation operators. This exploits the rich expressive power of SQL and most of the data manipulation in a CQL query is performed using standard SQL

constructs [8].

Other Functionality

STREAM has several other important pieces of functionality such as Operator Scheduling, Query Plan Optimisation and Synopsis sharing which are outside the scope of this review.

Discussion

STREAM is a stream processing system which extends the concepts used in traditional database systems. The ability to combine traditional stored relations with streams is particularly useful. The system also leverages existing relational semantics allowing the current techniques for optimising query plans to be used. The system also extends SQL reusing many of the constructs which are already familiar to programmers which should result in a reduced learning curve for new users of the system. The data model used in STREAM is not as conducive to modeling using events as the Aurora model. The tagged tuples used map well to events as do the deletion and insertion of relational data, however the mapping which follows from the r-stream operator which in this mapping creates a stream of events from a relation is less clear and does not clearly translate to any current actions in the event-based model. STREAM provides its own language CQL in which all its queries must be written. This has additional benefits in that creation and optimisation of query plans is made possible, however it does tie the application developer to a specific language. STREAM is designed for stream-monitoring applications and for answering continuous queries over these streams. STREAM is designed to answer queries across multiple streams which include sensor streams so it has good facilities for combining the output from multiple sensors. It provides numerous join, intersection and union operators for combining streams. STREAM also provides three types of sliding window which allow the application to handle continuous sensor data. The STREAM system however is limited by CQL in the expressiveness of the queries which it can answer. It is impossible to specify an arbitrary query in the manner of user-defined operators in the Aurora system. This shortcoming prevents the STREAM system from being capable of supporting sensor-driven applications. The STREAM system is resource-conscious and has facilities for load-shedding and query optimisation, however it is hard to conceive how its current design could be implemented on a resource-constrained device typical of the mid to lower range of the ubiquitous computing domain.

3.6 Stream and Event Processing Systems

Stream processing and event stream processing have similar ancestry, stream processing with its roots in database systems and event processing with its roots in event-condition-action rules in active databases [27, 29, 26, 49, 76, 139], so there is plenty of common ground between the two research areas. A number of systems exist which claim to perform both event and stream processing. These include Coral8 [66], StreamBase [73], Esper [67], RuleCore [72], Aleri [62], Amit [3] and MavEstream

[78].

MavEStream was chosen for review as it aims to combine event and stream processing and is a research project and so is well documented.

3.6.1 MavEStream

MavEstream [78, 28, 47] is a system which aims to synergistically combine event and stream processing. It proposes to achieve this by combining the MavStream [82, 51] stream processing system with the Local Event Detector (LED) [27] event processing system. MavEstream performs continuous queries over streams and uses the results of these queries to raise events which can then be composed into complex events by the event processing system. It proposes a semantic window which extends the current window concept for continuous queries to allow developer-defined windows based on criteria other than number of tuples, time etc. The MavEStream architecture is shown in figure 3.9.

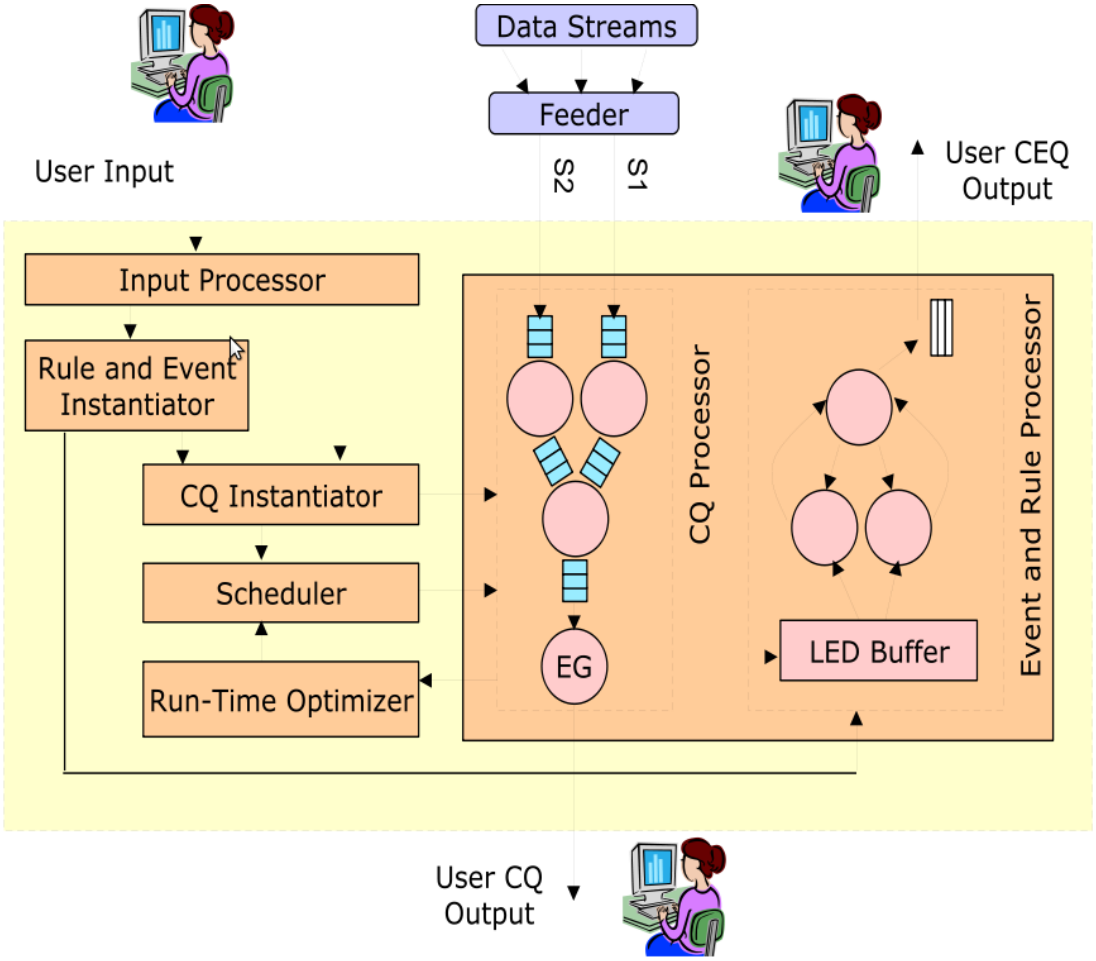

Figure 3.9: MavEStream Architecture [78]

Event Processing and Stream Processing Integration Model

MavEStream aims to integrate event processing and stream processing. Figure 3.10 shows the four stage integration model used. Stage one is the stream processing stage. The MavStream [51] stream

processing system is used to process incoming stream data. MavStream is a data stream management system designed for application areas such as security, telecommunications, manufacturing, pervasive environments, click stream analysis and military applications [51]. MavStream uses a client-server architecture in which the client accepts input queries from the user and sends the request to the server. The stream operators in MavStream are designed to produce continuous real-time output.

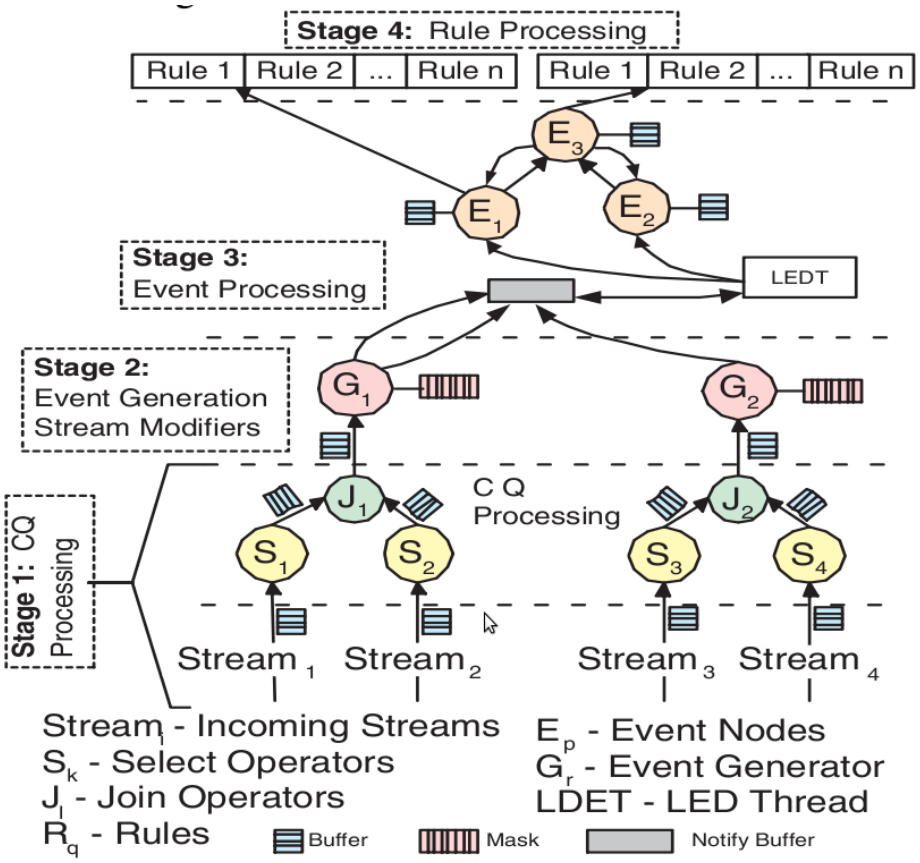

Figure 3.10: MavEStream four-stage integration model [78]

The stream processing stage accepts stream data as input and evaluates continuous queries over this stream data producing stream output. The MavStream stream processing system has a number of operators including Select, Project, Join, Aggregates(sum, count, max, min and average) and Group By. The output from the stream processing stage is then passed through the event generation stage to the event processing stage.

Simple events originate in the stream processing and event generation stages and composite events based on these events are detected in the event processing stage. The LED event processing architecture is used to detect composite events and propagate these events to the rule processing stage. The rule processing stage is also part of the LED and it processes rules that are associated with the simple and composite events detected in the event processing stage. The LED is based on the event-condition-action (ECA) paradigm. *Event Operators* are used to construct composite events. Some of the event operators supported include : OR, AND, Sequence, NOT, Aperiodic, Periodic, Plus and

Cumulative Aperiodic and Periodic. LED supports four event consumption modes: recent, chronicle, continuous and cumulative [29]. In the recent consumption mode only the most recent initiator of a composite event is used. In the chronicle consumption mode the oldest initiator is paired with the oldest terminator. In continuous mode a terminator event can be used with multiple initiators, i.e.,the terminating event can be consumed in multiple composite events. In the cumulative event mode all occurrences of an event type used in the detection are accumulated and discarded once a terminator for that event is detected.

Continuous Query Processing

The bottom layer of MavEStream is composed of a continuous query stage. Streams are used as inputs and continuous queries are evaluated over them. The output of this stage is an output stream. Continuous queries are named and the output streams can be used as the input stream to other continuous queries. Scheduling and QoS concerns are also addressed at the continuous query processing stage.

Event Processing

Users can specify events based on continuous queries. Event creation takes the form:

```
CREATE EVENT name
SELECT A1,A2,..,An
MASK Conditions
FROM Es | En
```

An event is created with the specified attributes, which satisfy the MASK, on the continuous query Es or event expression En. The event expression allows an event to be specified on two or more events.

Semantic Window

In standard stream processing systems windows are usually either tuple-based or time-based, with some systems allowing attribute-based windows. A semantic window is proposed in MavEStream which allows a computation to be made to decide what elements of the stream should be included in the window. This computation is executed and decides what tuples to add to the window and what tuples to remove from the window. This allows greater expression of window membership than what is traditionally available in data stream managements systems.

3.6.1.1 Discussion

MavEStream combines stream processing with event pattern detection. It supports this by allowing events to be produced as the output of continuous queries and these events to be combined into complex events according to a pattern supplied by the user. This model is a reasonably straightforward juxtaposition of the two paradigms that allows not only stream processing but complex event and rule processing. The semantic windows proposed are also a potentially useful addition to the field.

Apart from semantic windows MavEStream does not provide any new abstractions for combining sensor data or for analysing sensor streams. Its contribution is based around the realisation that both stream processing and event processing are needed in a wide array of applications. Because of this MavEStream suffers from the same strengths and ailments when it comes to sensor-driven applications as the event and stream processing efforts that constitute it. The system supports combining sensor streams and has abstractions for dealing with historical stream data, however custom analysis of sensor output from multiple sensors is not possible with MavEStream.

MavEStream is quite a resource hungry application. Effectively the LED and MavStream frameworks are running at the same time on the same machine. Therefore MavEStream would only be usable on platforms at the higher end of the platform spectrum shown in figure 2.1.

3.7 Sensor Database

The sensor database paradigm is a popular paradigm for building monitoring applications using wireless sensor networks. Using the approach a network of largely homogeneous sensor nodes is abstracted as a database that can be queried by the user. This significantly reduces the complexity that the developer must face and allows non-programmers e.g., biologists, geologists etc. to exploit wireless sensor networks in their research. There are a number of projects which treat a sensor network as a database in some manner including [138, 95, 126, 37]. We have chosen to review the TinyDB [95] and SINA [128] projects here as they are the most relevant and influential in the field and contain important abstractions for implementing sensor-based applications.

3.7.1 TinyDB

TinyDB [95] is an acquistional query processing system for sensor networks. It is designed to work on Berkeley motes and provides a query processing interface to the sensor network allowing users to specify queries in a similar manner to relational databases. A central design idea is the notion that significant power savings can be made in a sensor network by controlling the acquisition of data and that the query plan and processing system can control this process. The system is built on top of the TinyOS [60] operating system. Figure 3.11 shows a query in a tinyDB network. Users specify queries at a base-station which is then optimised and disseminated to nodes in the sensor-network. Nodes in the sensor-network sample the sensors intelligently with power conservation being an over-riding concern.

3.7.1.1 Query Language

TinyDB uses a SELECT-FROM-WHERE-GROUPBY clause supporting selection, join, projection, and aggregation [95, 94]. Sensors in the system are viewed as a single table with one column for each sensor type. The schema of the table is the same throughout the sensor network. The sensor table is unbounded and therefore may not be used with certain blocking operations such as sort and

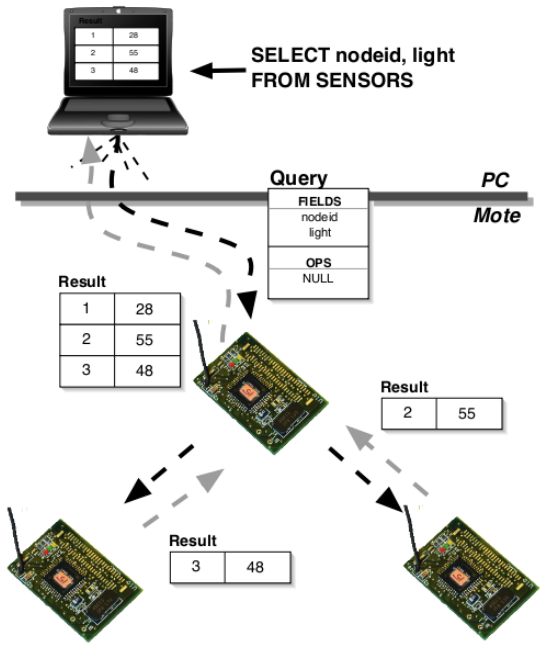

Figure 3.11: A query and results propogating through a TinyDB network [95]

symmetric join. A bounded subset of the stream or window must be specified or the sensors table can be the outer table in a nested inner-loop join.

The FROM clause can refer to stored tables and the sensor table. Stored tables are supported using materialization points which reside at a particular node. They can be used to support sub-queries and also for windowed operations, where a portion of a stream is stored at the node and queried later.

Data aggregation is a popular technique in sensor-networks to reduce the overall energy use by reducing the amount of data which is transmitted throughout the system. The cost of transmitting data is expensive in sensor-networks particularly the nodes near the route using a multi-hop communication protocol. Aggregation works by pushing part of the processing into the network and communication less higher-level data back to the root, rather than raw sensor data. TinyDB supports aggregation queries which are designed to reduce the amount of information that travels through the network by pushing the processing into the network. TinyDB also allows users to extend the system by authoring software modules which implement an aggregate. TinyDB users may extend TinyDB with user written aggregates by authoring software modules that implement the required functions [95]. These functions are a merging function, and initiliazer and an evaluator. They operate under the constraint that the merging function must be commutative and associative. TinyDB supports temporal aggregates and materialization points, both of which can be used to handle historical data, by creating structures which are similar to sliding-windows.

TinyOS is event based and TinyDB also uses events. Events can be used as a mechanism for initiating queries. Events in TinyDB are generated by another query or from the underlying operating system. This could for example come from a sensor attached to the node. TinyDB events are not

distributed throughout the network, just at the local node. Events in TinyDB allow the node to be dormant until some external event occurs, and are central to the notion of acquistional query processing. Events can also be used as stopping conditions for queries. Events can also be outputted as the result of a query on a node which along with materialization points provide support for a form of nested queries [95].

3.7.1.2 Discussion

TinyDB is a sensor database application for sensor networks. It is designed to simplify the process of acquiring sensor readings from sensor networks with the additional design goal of reducing power consumption. It is well suited to resource-constrained devices and it is aimed at reducing the power usage in such devices which is generally the most constrained resource in the sensor-network domain. TinyDB is highly effective at retrieving sensor readings and is a massive improvement for the developer over writing and testing applications in an ad-hoc manner. The declarative language also simplifies the process for the developer and removes the chore of rewriting low-level sensor code and allows the programmer to focus on writing sensor-data acquisition specific code. The declarative language facilitates optimisation and facilitates acquistional queries which result in reduced power consumption. TinyDB is well suited to sensor monitoring applications, however it is not designed for sensor-driven applications where the application which is being driven resides in the network? One severe limitation of the TinyDB approach and the sensor-database approach in general is that it makes an assumption the the sensor nodes are Homogeneous. This works well for a relatively unsophisticated network, but as soon as we introduce additional sensors and sensor types it becomes unmanageable.

While TinyDB supports events as the initiator and potential outputs of its queries it is intrinsically tied to the TinyOS implementation. The events limitation of only being delivered on the local node underlines how the system is not designed to use events in the traditional sense, but purely as the communication mechanism available in TinyOS.

TinyDB uses a custom language. This language provides additional benefits in the form of query optimisation and additional constructs for purposes such as specifying sampling rate and data aggregation. It is difficult to see how this would map to an alternative language.

TinyDB is well suited to applications that require simple monitoring and aggregation of sensor data in a sensor-network, particularly ones which are under power constraints. However this corresponds to a small section of the application space in ubiquitous computing and neglects sensor-driven applications which we see as particularly important for the future of sensor-rich ubiquitous computing applications.

While TinyDB allows programmers to collect sensor data, combining this data is not well supported in TinyDB. The aggregation function performs a form of sensor fusion on the data but it is limited to operations on a single sensor, e.g., the average of a particular sensor in the system. Performing analysis on two sensor streams is not well supported in TinyDB.

TinyDB does support continuous historical data through the use of materialization points, which allow the programmer to store readings in a table and use this table in subsequent queries. For

example this allows the developer select the max or average of a certain number of readings which are stored in the table.

TinyDB does not support advanced functionality for combining and analysing sensor readings from multiple sensors using historical data.

3.7.2 SINA

The Sensor Information Networking Architecture (SINA) [126, 128, 77] is an architecture that facilitates querying, monitoring and tasking of sensor networks. It is designed to fit the needs of a wide range of sensor network applications from military to factory automation. SINA acts as a middleware layer providing a set of configuration and communication primitives that enable scalable, energy efficient and robust interactions between sensor nodes [128]. To support scalability in the network SINA provides a feature called hierarchical clustering. Sensor nodes are aggregated to form clusters based on power levels and proximity, with a cluster head performing information filtering and aggregation of the clusters information. A clustering algorithm is provided to autonomously elect and re-elect cluster heads. Attribute-based naming is the preferred scheme in SINA. This allows applications to program the sensor network without referring to individual sensor nodes. For example the name *[type = temperature, location = NoE, temperature =103]* describes all the temperature sensors located in the northeast quadrant with a temperature of 103 [126].

3.7.2.1 Information Abstraction

The SINA information model views a sensor network as a collection of datasheets. A datasheet contains a collection of attributes of each sensor node. Each attribute in a datasheet is referred to as a cell and the collection of datasheets in the sensor network make up an associative spreadsheet. On initialisation of the sensor node a small number of cells exist, however as time progresses more cells can be created as a result of the operation of the applications in the network. A cell may contain a single value e.g., battery level or multiple values e.g., temperature changes in the last 30 minutes.

3.7.2.2 Sensor Query and Tasking Language

The sensor query and tasking language (SQTL) provides the programming interface between the sensor applications and the SINA middleware. It is a procedural scripting language which is designed to be flexible and compact which also supports simple declarative query statements. SQTL is designed to be similar to standard procedural languages but also has a lightweight object-oriented features [77]. The SQTL language includes constructs for arithmetic (+,-,*,/), comparison (==, !=, <, >), and Boolean (AND, OR, NOT) operators, assignments, conditional construct (if-then-else), loop construct (while), object instantiation (new), and event handling construct (upon) [77]. Access is provided to the sensor hardware using primitives such as *getTemperature* and *turnOn*. Communication primitives are also provided e.g., *tell* and *execute*. In addition to these primitives SQTL also provides an event handling construct. Three types (sources) of events are supported by SQTL. These are events generated by the

reception of a message at the sensor node, events generated periodically by a timer and events caused by the expiration of a timer. These events are defined by the SQTL keywords *receive*,*every* and *expire* respectively. By using the *upon* construct the application programmer can create an event handling block. SQTL scripts are intended to be executed by every member of the network however they can be targeted at specific receivers or groups of receivers using an SQTL wrapper which is constructed using XML.

3.7.2.3 Declarative Query Language

A built-in declarative query language is provided for applications that wish to collect sensor information. The query language is adapted from SQL and operates using the cell information model described above. The query

```
SELECT avg(getTemperature())
AS avgTemperature
FROM CLUSTER_MEMBERS
```

would ask every cell to create a new cell called avgTemperature which maintains the average temperature among its cluster members.

3.7.2.4 Discussion

SINA provides a number of interesting abstractions that could be used to create sensor-driven applications. The inclusion of a procedural language allows complex tasks to be specified (as opposed to the network being queries). The inclusion of the XML wrappers allows the application programmer to target the script at a particular node in the network, which overcomes the inherent weakness in other sensor networks, in that the application must be homogeneous across the network. The project also uses events and this allows the application to respond to messages from other nodes and interface with timers. Historical sensor readings are supported in the *associative spreadsheet* abstraction. It is possible to run a query that stores e.g., the last 30 readings in a cell of the spreadsheet and presumably query this at a later date. Each query could be considered to be a window over previous sensor readings, and depending on the windows specification different views over the historical sensor reading could be maintained.

However the procedural provided by SINA is very simple and may not have the sophistication for more complex applications. For example no mention is made in the literature of how to create or call a sub-routine or function in SQTL. Also while events exist in SINA and allow the application to react to incoming messages from other nodes, it could not be described as a complete event system. The system doesn't provide any mechanism to combine events although the procedural query language appears to have at least one construct (avg) for aggregation of sensor readings.

In summary SINA appears to be positioned somewhere between the homogeneity of the nodes in TinyDB and the configurability of programing nodes independently. The inclusion of historical sensor reading mechanism could help programmer write applications dealing with continuous sensor

Name	Event Model	Programming Languages	Platform	Application	Combination of Sensor Data	Continuous Sensor Data	Analysis of Continuous Sensor Data
Java Beans Event Model	O	X	O	O	X	X	X
STEAM	O	O	O	O	X	X	X
Cambridge Event Architecture	O	O	X	O	O	O	X
Composite Event Detection as a Generic Middleware Extension	O	O	X	O	O	X	X
SASE	O	X	X	X	O	O	X
Cayuga	O	X	X	X	O	O	X
Aurora	X	X	X	X	O	O	O
STREAM	X	X	X	X	O	O	X
MavEStream	O	X	X	X	O	O	X
TinyDB	X	X	O	X	O	O	X
SINA	X	X	O	X	O	O	X

Table 3.1: Summary of State of the Art

streams. The inclusion of a procedural language for specifying the tasks highlights the inadequacies of a declarative language for the job. However no abstractions exist in the language that aid the combination of sensor readings that do not exist for a person writing the applications in a procedural language such as C and polling the sensors for data.

3.8 Summary

There is a large body of work related to event based systems and stream processing systems. While to two research fields are evolving separately there is a large amount of common ground between the two. Both fields have their origins in the database community and both have evolved to meet different needs. With the rapid increase in the number of small cheap sensors and the need to combine this data in meaningful ways the two field's paths are converging again.

The event based programming model is the programming model of choice for developers of ubiquitous computing applications. There are a large number of different event systems and we have reviewed several ones that are important to the thesis in this chapter. Stream processing is also a large domain and we have analysed two important research projects from this domain. The sensor

database abstraction is popular in sensor networks and provides a mechanism to abstract above the individual nodes in the network. Both sensor network approaches reviewed used events to some degree and had support for historical data illustrating the requirements for this functionality in sensor based applications. MavEStream is a very interesting project which has similar goals to this thesis although with different focuses and attempting to fuse the two models so that the system has both event composition and stream processing. MavEstream also does not provide abstractions to allow custom code to access historical sensor data.

It is clear from the analysis of the work that while abstractions to combine streams of data are commonplace in both event programming and stream processing, allowing this process to execute custom code is lacking in most projects, with the exception of Aurora which supports custom operators. The concept of historical event data is lacking in the event literature and this functionality is crucial to implement more advanced applications as is shown by its inclusion in the complex event applications and the stream processing literature. Exposing this functionality to the application developer would allow the programmer to write more complex applications of the type described in chapter 2.

Table 3.1 summarises the projects reviewed in this chapter. It details the requirements of sensor-driven applications that the applications meet. To effectively display the data in a binary format, it is necessary to approximate the information. Therefore an optimistic approach is taken, if the project goes any towards meeting the requirement it is considered to have met it. As we can see from the table the event-based systems meet many of the general middleware requirements of sensor-driven applications. Also we can see that the stream and event stream systems generally meet two of the three main functional requirements of sensor-driven applications to some degree. We can see that Aurora meets all of the functional requirements (analysis of continuous sensor data to a lesser extent) but fails to meet any of the general middleware requirements of sensor-driven application. We can clearly see that there is a gap in the state of the art where no system meets all of the requirements, both general middleware requirements and functional requirements of sensor-driven applications. This is the gap that the AESOP abstractions presented in this thesis address.

Chapter 4

The AESOP Model

This chapter describes the Architecture for Fusing Events using Streams and Execution Policies (AESOP), an architecture that can be used to add support for analysing continuous event data and analysing events from multiple sources to any event system that satisfies a minimum set of requirements. Because AESOP has a minimal set of requirements of the event model it can be used to extend a large amount of pre-existing event-models. This has two significant advantages. Firstly, there is a large amount of theoretical work already performed in the creation of event-models and analysing the characteristics of these event models, e.g. [124, 15, 3, 54, 79, 87, 100, 122, 129]. This work can be leveraged by extending these event-models using AESOP. Secondly, there is a large number of event models, e.g., [112, 57, 119, 92, 114, 56, 104, 103, 99] that have already been implemented and are already developed, tested and being used in the software industry. The AESOP model can be used to extend these event systems, allowing pre-existing implementations be re-used, saving significant development time and effort.

This chapter describes the design of the AESOP architecture. Chapter 5 presents two instantiations of the architecture, which are used to evaluate the architecture in chapter 6.

In section 4.1 we discuss the set of requirements that must be satisfied by the host event model in order for it to be suitable for extension by AESOP. Section 4.2 gives an overview of the AESOP architecture, discussing its interaction with the host event system and the effect AESOP has on the functional and non-functional characteristics of the host event system. It also discusses the interaction of the three major abstractions provided by AESOP and the interaction between AESOP and the application. In section 4.3 the event stream abstraction is introduced along with the windowing mechanisms that are used in AESOP to support historical event data. Section 4.4 introduces multi-event handlers which support analysis over multiple event streams. Section 4.5 describes execution policies which are used to detect event patterns in the event streams in order to determine when to execute the multi-event handlers. Section 4.6 shows how multi-event handlers can be composed to build more complex systems and section 6.7 summaries this chapter.

4.1 Requirements

AESOP is designed as a generic extension for event models. There are a minimal set of requirements that an event model must satisfy to be suitable for extension.

1. Events in the event system must be able to be categorised into groups, usually this is achieved by having an event type or subject.
2. It must be possible to subscribe or register interest in these event groupings.
3. It must be possible to publish events.

Various event models have differing time models. Some use a total temporal ordering on events, in others systems this is not desirable or possible. The AESOP model makes no demands about the timing model of the underlying system.

In chapter 3 we reviewed a large number of event systems. All of the systems from chapter 3, which use an event model, meet the set of requirements presented here. The requirements presented here are very general and are applicable to the overwhelming majority of event systems intended to be suitable for developing event based applications. Of the event systems which allow programmers to publish events, and are not therefore just one-way communication mechanisms, the author has yet to encounter an event system which fails to meet the remainder of the requirements.

Event systems can use different criteria to decide where the boundaries are between different events and what events to send to each consumer. For example some event systems can be typed, others are semantic and others content based. All event systems share a common characteristic in that they do classify events for delivery to consumers. AESOP relies on this categorisation of events and is agnostic to how it is performed.

4.2 Overview of the AESOP Architecture

The AESOP architecture offers three core abstractions: *multi-event handlers*, *event streams* and *execution policies*. The multi-event handlers are functions that are executed in response to the arrival of one or more events. The execution policies are expressions that are used to decide what combination of incoming events trigger the execution of the multi-event handler. The event streams are streams of data used to supply the multi-event handler with event data and to facilitate analysis of historical sensor data.

4.2.1 AESOP and the Host Event System

The AESOP architecture as shown in figure 4.1 is positioned between the host event system and the application. AESOP is designed to be usable to extend any event-based system. The assumptions it makes are very general and are simply that the system allows AESOP to publish events, subscribe to events and be notified when such events happen. Figure 4.2 shows the interface between AESOP

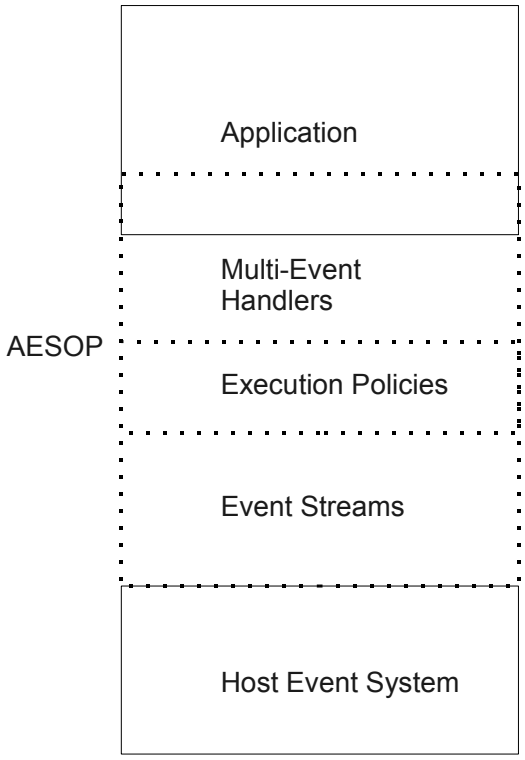

Figure 4.1: Placement of the AESOP Architecture

and the host event based system. From the figure we can see that the AESOP model is separate from the host event system and communicates with it entirely using the standard interface for subscribing to, publishing and receiving event notifications. The event streams subscribe to and are notified of events from the host event system and the multi-event handler can publish events with the host event system.

4.2.2 Feature Preservation

As discussed in section 1.2 and section 3.2 there are a large variety of event systems which have wide range of functional and non-functional features that are designed to address a range of challenges. These features - such as distribution, scalability and proximity and content filtering - are reasons used to chose a particular event system as they are designed to address shortcomings in event systems or to solve a particular problem. The AESOP architecture preserves the functional and non-functional features of the event system that it extends. Therefore a distributed event system which is extended with AESOP continues to be a distributed event system. AESOP however does not provide distribution to an event system, and a non-distributed event system will remain non-distributed. The AESOP architecture adds its features and abstractions to the features and abstractions of the host event system, allowing the programmer to use both in the implementation of their application.

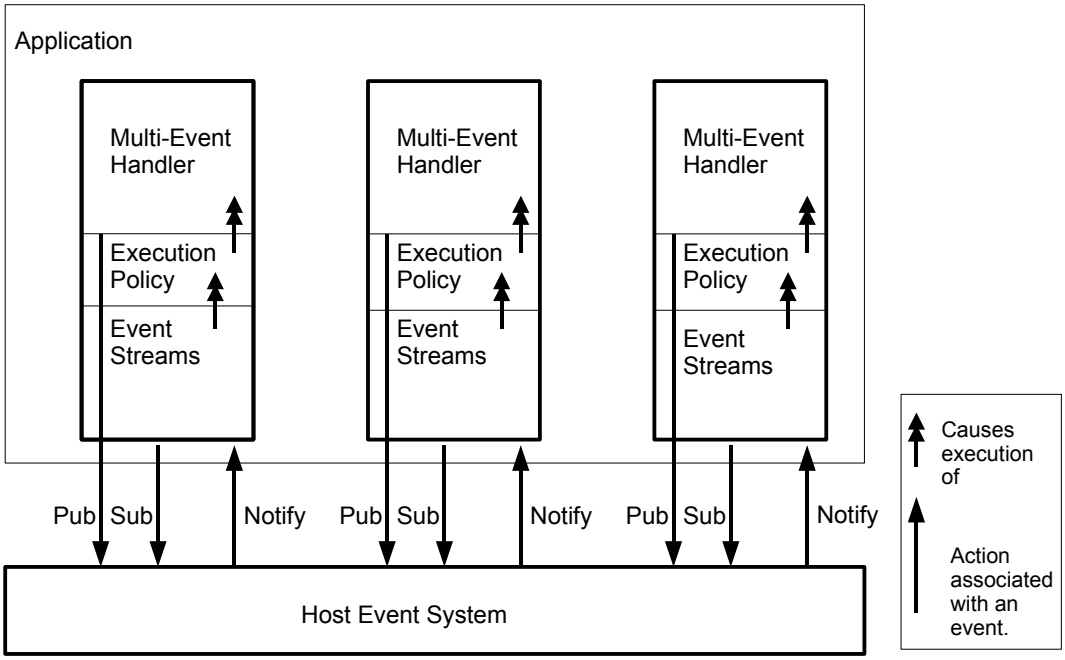

Figure 4.2: AESOP Architecture Overview

This preservation of features and the design of the AESOP architecture as a generic extension for event systems allows a large body of work both theoretical and practical to be leveraged.

4.2.3 Interaction between AESOP and the Application

An overview of the AESOP architecture is shown in figure 4.2. We can see that there is an overlap between the multi-event handlers and what would be considered the application code of the system. The code that gets executed in the multi-event handlers contains application-specific code. This code performs some analysis of the events in its event stream windows and can raise new events. It can also perform standard application functions in the same way that application code in traditional event handlers, e.g., in STEAM [99] or JavaBeans [103] can be used to implement application logic in traditional event based programming.

4.2.4 Interaction Between the AESOP Abstractions

Figure 4.2 shows the interaction between the AESOP architecture, the application and the host event system. It also shows the interactions between the three abstractions in the architecture and the application and host event system. The event streams are the main interface to the host event system. They receive events that are published and are of the type specified in the event stream. The multi-event handlers also interact with the host event system because they can raise events as a result of their execution.

Execution policies and event streams also interact; execution policies evaluate when a new event arrives in a window of an event-stream of the multi-event handler which it is associated with. This is shown in figure 4.2 with the double arrow notation between the event streams and the execution policy. If the execution policy evaluates to true then the multi-event handler gets executed. This interaction is also shown using double arrow notation in figure 4.2. The multi-event handler can read the values of all the events in the windows of the event streams which are attached to the multi-event handler, and as stated earlier can then raise events using the host event system.

4.3 Event Streams

An *event stream* in AESOP is a continuous sequence of individual events of a specific type. Event streams are the abstraction which facilitates all communication between multi-event handlers in the programming model. They serve as input streams and output streams to multi-event handlers. Event streams are the basic abstraction for handling continuous sequences of events.

Event streams are ordered by arrival time in AESOP, with events that arrived latest higher in the sequence. This is independent of any timestamp or other attribute that the event may posses. Events which arrive simultaneously in the AESOP system are added to the event streams in the order that they are processed. Because event streams are continuous they can potentially grow to infinite length. Event streams are not shared between multi-event handlers and there is no requirement to synchronise event streams across multiple nodes in a distributed event-based system.

When an event stream is created in AESOP it can be initialised with a sequence of events or remain empty.

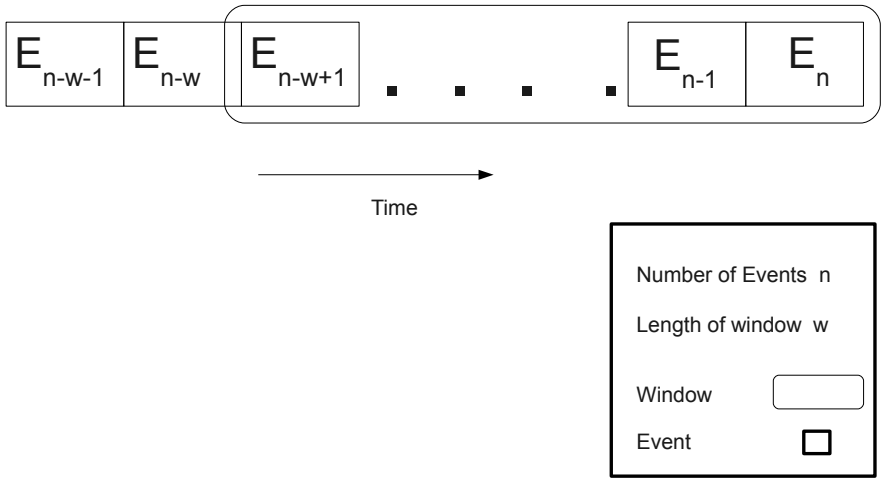

Figure 4.3: Event Stream Showing Window

4.3.1 Event Stream Windows

Event streams are unbounded streams of event data and can potentially be infinite. Dealing with this vast quantity of data is one of the challenges of writing sensor-driven applications. In order to make relevant data from event streams available to the programmer we introduce the concept of a window over an event stream, a concept that is commonly used in many streaming systems and complex event processing systems in order to handle historical data, e.g., [8, 2, 54, 41, 78]. A window allows a section of the event stream to be retained and viewed by the multi-event handler. The section that is retained is the part of the event stream which is relevant to the multi-event handler, events can be excluded from the window, however the window maintains the same ordering as events in the event stream. Figure 4.3 shows a window over an event stream. Events outside of the window are discarded and are no longer accessible from this window. Events inside the window are accessible for as long as they remain in the window. Event stream windows have both window mechanisms and window membership schemes. Window mechanisms determine the relationship between successive views of the window and are discussed in section 4.3.2. Windows membership schemes determine which of the individual events in the event stream are members of the window and are discussed in section 4.3.3.

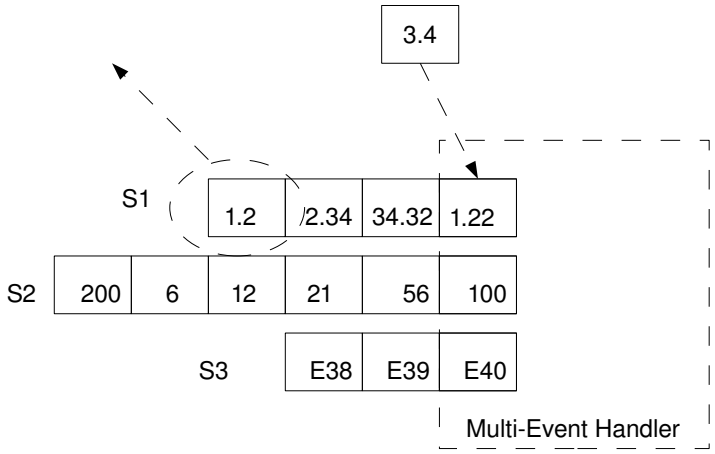

Figure 4.4: Event Streams and Sliding Windows

4.3.2 Windowing Mechanisms

Associated with each input stream to a multi-event handler is a window. This window is used to decide what and how many previous events should be stored in order to satisfy later analysis on the stream. There are two main *windowing mechanisms* supported by the AESOP model: *sliding windows* and *tumbling windows*.

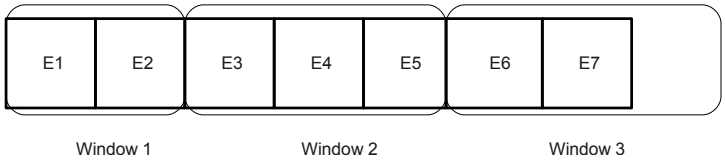

Figure 4.5: Tumbling Windows

Sliding Windows

Sliding windows store a fixed number of events defined when the window is created. They have a fixed maximum length and once they are full, the arrival of new events in the window pushes the oldest event from the window. Sliding windows are a type of window mechanism in which subsequent views of the windows share events. The difference between the nth and the $(n+1)th$ view of a particular window would differ by a maximum of two events, the oldest event that had left the nth view and the newest event that had just joined the $(n+1)th$ view. Figure 4.4 shows an example of some sliding windows on event streams. In this example we have three event streams. The first event stream S1 has events with float data, S2 has events with integer data and S3 is shown as having events with an arbitrary type. The lengths of their sliding windows which are specified when initialising the event streams are 4, 6 and 3 respectively. As new events arrive they join the head of the queue pushing the last element out of the sliding windows range. In the example we can see an event with data 3.4 is arriving at the front of S1 pushing the event with value 1.2 out of the sliding window.

Tumbling Windows

Tumbling windows are windows which keep accumulating events on an event stream until a tumbling condition is evaluated which causes them to discard all their contents and begin accumulating events again. There are no common events between the window pre-tumble and post-tumble. If for example the multi-event handler was analysing events on an hourly basis, a tumbling window could be used. At the beginning of the hour the window would be empty and it would gradually fill up over the course of the hour as new events arrived. At the end of the hour, it would "tumble" and empty and begin filling up again. The same tumbling mechanism can be used on any attribute of the event. To do so the developer must specify a condition on which the window should "tumble". Figure 4.5 shows a number of consecutive views of a tumbling window on an event stream in AESOP. We can see that the first view of the window contains events $E1$ and $E2$ while the second view of the window contains $E3$, $E4$ and $E5$. The third view of the window contains $E6$ and $E7$ and has still not tumbled. The *tumbling condition* is not specified here, but in general it can be based on an attribute of the window, e.g., length of the window or on attributes of the events, e.g., a timestamp.

4.3.3 Window Membership

Every window on an event stream needs a window mechanism and a method for determining membership of the window. Window membership in the AESOP architecture ranges in complexity from allowing all events in an event stream to be members to arbitrarily complex expressions using the attributes of the events to determine exactly what events belong in a particular window. We call the condition that evaluates to determine if an event can have membership of a particular window the window's *window membership expression*. The window membership expression is used in combination with the window mechanism to determine what events are in the most recent view of the window at any time. Therefore a window membership expression could evaluate to true for a given event, however if the windowing mechanism is a sliding window with a length of 3, and it is the fourth oldest event that satisfies the window membership expression for that particular window, it will not be viewable in the window.

Window membership expressions are evaluated when an event arrives on the event stream of the window. There are two types of window membership expressions. *Arrival* window membership expressions evaluate only for a particular event when that event arrives in the event stream. *Updating* window membership expressions evaluate when any new event arrives on the event stream. Arrival window membership expressions are used when the relative values of the event's attributes remain constant with time and the execution of the application. Updating window membership expressions are used when the attribute's relative value is changing and the window membership is changing with it. The main example of a window particularly suited to being an updating window is when implementing a time-based window. If the events had a timestamp attribute and the window membership expression was using this timestamp to restrict window membership to a certain number of minutes in the past, the window membership expression would need to be re-evaluated whenever new event data arrives in the window to ensure that the window membership remained current.

Not all incoming events on an event stream can cause the execution of the multi-event handler. It is possible for an event to arrive in an event stream and not meet the membership requirements for a window. This incoming event would not be added to the window and would not cause the execution of the multi-event handler. The newest event in an event stream window is passed to the execution policy on arrival and is used by the execution policy to decide if the multi-event handler should be executed. The present contents of all the windows associated with a multi-event handler are available to the multi-event handler when it gets executed. This provides a mechanism by which the multi-event handler can analyse previous event data and can deal with the continuous nature of streams of events.

4.3.4 Instantiation Considerations

There are a number of programming abstractions related to event streams that must be mapped in any concrete instantiation of the AESOP architecture. The structure of the abstractions in the target programming language depend on the programming language and the programming methodologies

it supports. However we can clearly identify the important abstractions and highlight the important decisions that need to be made when instantiating the architecture. The mechanism for creating event streams is important. What window mechanisms are supported and how they are specified by the developer is also very important. The degree of expressiveness of window expressions will determine how they are used and will significantly influence any applications developed using the instantiation. This is also true of the tumble expressions.

The various windows supported in the model are made available to the developer in the multi-event handlers and may be available in the execution policies. The manner in which the data structure is exposed to the developer is a very important consideration. Developers will spend a lot of time accessing the event stream data through these abstractions so they should be as close as possible to the standard mechanisms used in the target language.

It would be possible to design a specific language which could be used by the application developer to specify window membership. However, this may negatively impact on the resource usage of the final application and also would create an additional barrier which would make it harder for developers to begin using the abstractions.

4.3.5 Event Streams in Example Application

In order to facilitate the description of the abstractions we use a simple example application which will be used for purpose of illustration in the discussion. The example we will use is the patient fall detection application introduced in section 2.1.2. This application uses multiple accelerometer sensors to analyse the posture of the patient. If the patient's posture has changed from vertical to horizontal, the application analysis the accelerometer data and determines if the change was violent and may be the result of a fall, or if it was characteristic of a sitting or reclining motion. We will use this example application to illustrate the AESOP abstractions as we describe them in the remainder of this chapter.

In our example application there are three event streams. Each of the three accelerometers output events, and these events constitute an event stream for each sensor. The application needs to be able to analyse the events from the accelerometers once it detects a change in posture of the patient. Therefore a window must be defined over each of the event streams. Sliding windows would be a suitable window mechanism, and window membership would be defined as the n most recent events, where n is the number of accelerometer readings required to properly recognise a violent fall.

4.4 Multi-Event Handlers

Multi-Event Handlers are code fragments that get executed in response to new events in the windows of one or more of the event streams associated with them. The multi-event handler has access to these windows while executing and contains logic to analyse the data in these windows. They can output new events and/or can perform application specific actions.

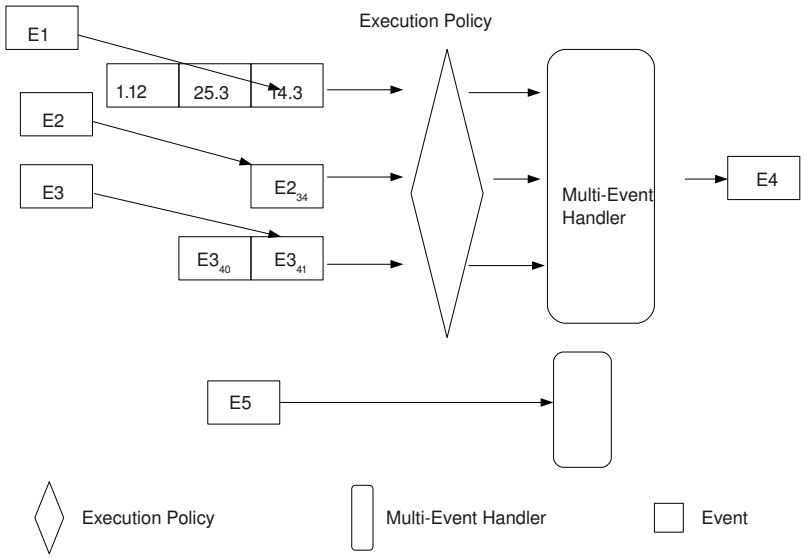

Figure 4.6: The AESOP Model, Multiple Events with Execution Policy

Multi-event handlers can be considered a generalisation of callbacks for traditional events. Figure 4.6 shows a traditional event handler, which executes when one event arrives, and a multiple-event handler in the same diagram. Multi-event handlers generalise the behavior of the traditional event handler and execute in response to multiple events. A traditional event handler is identical to a multi-event handler with a single input event stream that has access to a sliding window of length one on its input event stream.

The combination of input events that can cause the multi-event handler to execute is determined by an execution policy which is associated with the multi-event handler. Each multi-event handler has one or more event stream windows and exactly one execution policy and can output events in one event stream.

Multi-Event handlers can output the results of their execution as events which can form an event stream. This output stream can be used as the input to other multi-event handlers.

Multi-Event handlers have access to windows specified over the event streams and can perform analysis of the historical data in the streams. Because they have access to all the windows over all of the event streams when they are executed, it is possible to analyse historical event data over multiple event streams. In section 2.3 we identified this functionality as critical for many sensor-driven applications.

4.4.1 Instantiation Considerations

Multi-Event handlers like the other AESOP abstractions must be mapped into a concrete instantiation. Multi-Event handlers are the central abstraction in the architecture so their mapping is very important and significantly influences the structure of all applications developed using the architecture. A large amount of application code will be written in the multi-event handlers so it should be

clear and easy to understand and should map to the standard abstractions for developing applications in the chosen language, e.g. a multi-event handler could map to a method in an Object-Oriented language or to a function in a procedural language.

Listing 4.1: Pseudo-code for Multi-Event Handler in Fall Detection Application

```
1  meh_detect_fall{
2      inputs:
3      event_stream_window_a1,event_stream_window_a2,event_stream_window_a3;

5      //simple example algorithm if magnitude of acceleration > 15 the fall is
           violent

7      for( accel = event_stream_window_a1.first; accel != null;  accel =
           event_stream_window_a1.next){
8          if(magnitude(accel) > 15){
9              call_ambulance();
10             return;
11         }
12     }
13     //repeat for other event streams
14 }
```

4.4.2 Multi-Event Handler in Example Application

In the example pervasive healthcare application the multi-event handler would analyse the three accelerometer event streams and detect if the change in posture of the patient was due to a violent fall, and if so would proceed to notify the relevant people. The multi-event handler needs to be able to analyse the event stream windows from the accelerometers and use the analysis of this historical event data leading up to the change in posture to decide if a violent fall has happened. From a programming point of view this requires that the multi-event handler has access to a data structure that represents the historical data in the event stream. The multi-event handler must also have access to communication facilities so that it can send an alert if a fall is detected. Listing 4.1 shows a pseudo-code multi-event handler for the example described here. The algorithm used is very simple it checks the magnitude of the previous accelerometer readings to determine if they exceeded a threshold. In practise this algorithm would be much more complex taking into account the body position of the accelerometers and a model of the human body, to properly analyse the movement of the patient.

4.5 Execution Policies

Execution policies govern the way in which multi-event handlers decide when to execute given new event stream window data. In order for a multi-event handler to execute it needs to have valid data for all of its inputs. This means there must be at least one element in each of the windows of the input streams. This data can be initial event stream data or real event data. When new data arrives in one of the streams there is a multi-event handler specific decision to be made. We can decide to execute the multi-event handler using this new event or wait until all streams have new events, or some combination of the two options. Execution policies are the abstraction used to represent these decisions in the architecture.

Execution policies are expressions that are used to decide if the arrival of new data in an event stream window should cause the execution of the multi-event handler it is connected to. They are evaluated whenever new data appears in a window for the multi-event handler. They have access to all the windows on the event streams connected to the multi-event handler and can also access the attributes of the events.

The execution policy allows us to fine tune the control we have over the influence of the arriving data on the application. In the traditional event model, whenever an event is received the corresponding event handler executes. So, it is equivalent in this respect to having a multi-event handler with one event stream with an execution policy that executes whenever a new event arrives. As soon as we add the concept of additional event streams which may trigger the execution of the multi-event handler to the model we need the abstraction of execution policies to allow the application developer to specify exactly what combination of events will cause the execution of the multi-event handler.

With execution policies we can use arbitrarily complex expressions which specify exactly when we want our multi-event handlers to be executed. For example, if $S1$,$S2$ and$S3$ represented the presence of new events on stream1,stream2 and stream3 respectively, $S1 \wedge S2$ specifies that both $S1$ and $S2$ must have data in order to execute the multi-event handler. If we only need one of sensor $S2$ or sensor $S3$ to execute the handler, $S1 \wedge (S2 \vee S3)$ specifies that when new data arrives for stream1 and we have new data from stream2 or stream3 we execute the handler. The execution policies have access to the windows of all the event streams connected to the multi-event handler and all of the events in these windows can be used to decide if the multi-event handler should be executed. The attributes of these events may also be used in the execution policy expressions. This can be used for example to access timestamps and control the execution of the multi-event handler in this fashion. This high degree of control in the execution of the multi-event handlers allows complicated applications to be written easily using the abstractions.

Execution policies are used to simplify the multi-event handlers by separating the detection of relevant events from the performance of the analysis of the historical event data at these events.

Listing 4.2: Pseudo-code for Execution Policy in Fall Detection Application

```
1 accel_vector= hip_accelerometer_window.latest_event();
2 upright_accel_vector = {0,-9.8,0}
3     if (cross_product(accel_vector,upright_accel_vector) > threshold){
4         if(cross_product(hip_accelerometer_window.previous_event(),
              upright_accel_vector) < threshold){
5             return true; //posture has changed from vertical to horizontal
6         }
7     }
8 return false; // no change in posture
```

4.5.1 Instantiation Considerations

Execution policies contain a number of abstractions that must be mapped to a target language when creating a concrete instantiation of the AESOP architecture. How the execution policy is specified in the target language directly influences how expressive the execution policy can be. This directly influences the structure of the applications built using the instantiation. For example, valid implementations of execution policies range from simply specifying event windows which when they get new data we should execute the handler, to expressions which have the full expressiveness of a programing language such as C. Having less expressive execution policies should simplify the specification and testing of the execution policies and may be sufficient for some applications. Alternatively more complex expressions may be required for more complex applications.

It would be possible to design a language specifically for specifying execution policies for the AESOP model. However, as with window membership specification in section 4.3.2, this may negatively impact on the performance characteristics of the final model and would create a barrier that may prevent developers from using the model.

4.5.2 Execution Policy in Example Application

Returning to the fall detection application, the execution policy must detect a change in posture of the patient. It can do this by analysing one of the accelerometers, the one situated on the patients hip, and determining if the direction of the acceleration vector from the indicates that the patient is standing or horizontal. Listing 4.2 shows pseudo-code for an execution policy to detect a change in the posture of the patient. It determines the angle between acceleration vector of the patient standing upright and the present acceleration vector. If this is above a threshold it then checks the previous reading from the window, to determine if this is a change that has just taken place. If it is it returns true, causing the multi-event handler to be executed. If not it returns false and the multi-event handler does not get executed.

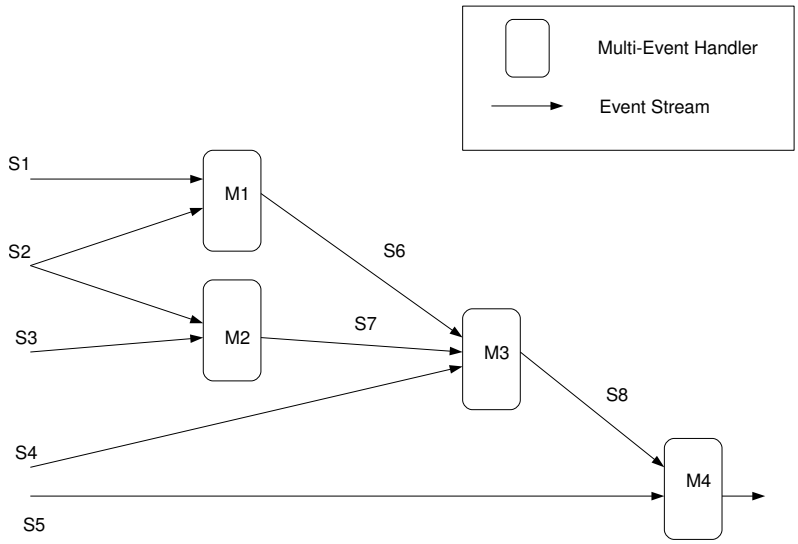

Figure 4.7: Composition of Multi-Event Handlers

4.6 Composition of Multi-Event Handlers

The ability to compose multi-event handlers into chains is a very useful and elegant feature of the AESOP model. The output of multi-event handlers are themselves event streams and can become the input to a second layer of multi-event handlers and so on. This is demonstrated in figure 4.7. Three layers of multi-event handlers are shown. The event data is being combined and analysed, creating higher level event streams such as S6, S7 and S8 which in turn can be used as inputs to other multi-event handlers.

The ability to compose multi-event handlers allows more complex systems to be built from simpler components. This both supports reuse of multi-event handlers, simplifies design of the system and simplifies testing. In order to compose multi-event handlers the host event system is used to distribute the events and the output from the multi-event handler is published back through the system. This also allows intermediate results to be known and used throughout the system. In distributed event based systems it also allows more complicated multi-event handlers to be distributed in the system and could for example allow a portion of the entire processing that needed to be performed to be positioned in the network near a particular source of events or on a node that was more resource rich than its neighbours.

It is possible when composing multi-event handlers, that composite events, which, by virtue of travelling through a number of multi-event handlers, may be older than newly arriving low level events. Depending on the application this could have a detrimental impact on the correctness of the resultant events and the application. There are a number of ways this can be addressed including waiting for event stability [87] and delaying delivery of the events. The decision on what scheme to adopt, if any, is heavily dependent on the features of the host event model, e.g., time model and distribution, and is left to the implementation of the architecture.

4.7 Summary

In this chapter we discussed the requirements AESOP makes of the host event system and the interaction between the host event system and the AESOP architecture. We described how the AESOP abstractions interact with the host event system and the demands they place on it. We introduced the three core abstractions in the AESOP architecture. We described these abstractions in depth and discussed the instantiation considerations that must be overcome in order to instantiate the AESOP model. We also described the interaction between the abstractions in the architecture. The three main abstractions are integrated in a final model and interact heavily with each other. We also discussed possible mappings of the abstractions that would be required in any concrete instantiation of the architecture.

Chapter 5

Implementation

In chapter 4 we described the abstract AESOP architecture. In order for the architecture to be used to develop applications it needs to be instantiated. Because of the range of possible host event systems and the large number of implementation decisions that can be made there are a large number of possible instantiations of the model. This chapter presents two particular instantiations of the AESOP architecture. We use these instantiations in chapter 6 to evaluate the architecture. We also discuss the major implementation decisions that need to be addressed when creating a concrete instantiation of AESOP. We describe how these decisions influence the functionality of the final extended event system. The major implementation decisions are presented to enable developers to extend existing event systems of their choosing with the functionality provided by AESOP.

The two instantiations of AESOP detailed in this chapter demonstrate how to extend an event system with the AESOP abstractions. They are therefore a sizable portion of the contribution of this thesis.

This chapter proceeds as follows, section 5.1 discusses the major considerations that any instantiation of AESOP must address. Sections 5.2, 5.3 and 5.4 describe the two instantiations of AESOP, C-AESOP and J-AESOP, and discuss how the AESOP abstractions map into the target languages. Finally, section 5.5 summarises the achievements of this chapter.

5.1 Implementation Considerations

Whenever we implement an abstract architecture there are always implementation decisions that need to be made depending on target environment. The AESOP architecture is no different and the following implementation considerations strongly influence the characteristics of the final implementation. These considerations are mentioned throughout chapter 4, we list them here and discuss them in the subsequent sections.

- The choice of host event system influences the functionality and behaviour of the subsequent augmented event system.

- The time model of the host event model is also important especially from the point of view of composition of multi-event handlers.
- The level of expressiveness of the execution policies in the instantiation directly influences the structure of the resulting applications.
- The set of windowing mechanisms that are implemented and the expressiveness of the window membership expressions influences the structure of the applications that can subsequently be developed using the instantiation.
- The abstractions presented are programming abstractions and there are a number of decisions to be made in how they map to the programming constructs and styles of the target language.

5.1.1 Choice of Host Event System

The choice of what event system to extend is one of the most important decisions when implementing the AESOP architecture. The host event system must satisfy the minimal set of requirements outlined in section 4.1. The functional and non-functional characteristics of the resulting extended event system will depend on the functional and non-functional characteristics of the host event system and so the host event system should be chosen carefully. For example, if we wish to create an AESOP implementation that is distributed, we must first decide on a distributed event system to extend. Similarly if we desire an event system that guarantees delivery of events or provides high scalability we must chose a suitable host event system.

The time model of the host event system is also important for AESOP. There are two inter-related points where there is a decision based on timing that must be addressed. Firstly whenever a new event is created in AESOP based on the combination of two or more events and both of these input events have a timestamp we must decide what timestamp if any to give the resulting new event. Normally we would assign the new event the timestamp of the youngest event which caused its creation.

The second timing question that must be answered when implementing the AESOP architecture is what decisions to make when combining events created as a result of previous events, with events that are potentially younger. If the event model does not support timestamps then the events can only be combined as they arrive. However if the event model supports timestamps we can address this issue. There are a number of techniques used by systems which combine events to tackle this problem. The Cayuga system reviewed in section 3.4 uses epoch based timing. All events with a particular timestamp are processed sequentially. Events that are created in this stage will have the same timestamp as the other events in the present epoch and will be processed before advancing to the next epoch. This scheme, while it may increase latency for some events, will solve the problem for a centralised system. In a distributed event-based system, coordinating the epochs across the many nodes is a non-trivial task. A best-effort approach to combining events will provide the simplest and fastest solution to the problem and is suitable for a system which is tolerant of dropped events.

The notion of event stability proposed in [16] can be extended to deal with delays due to the

execution of multi-event handlers. Event stability is a scheme used to deal with out of order events in distributed composite event detection. The composite event detection waits for the next event on an event stream before processing the previous one, to ensure that the correct event is being processed. This scheme could be modified such that multi-event handlers only process events older than events in a timing stream. Events which arrive on the other streams in the multi-event handler that are younger than the timing stream are not delivered to the event stream until a younger event arrives in the timing stream.

5.1.2 Window Mechanisms

There are two windowing mechanisms provided in the AESOP architecture. Not all implementations of AESOP will require both of these mechanisms. Which mechanism to support depends largely on the applications that the implementation is intended to support and what their requirements are. Some applications may require tumbling windows others may require sliding windows and others may require both. Applications which need to reason about all events that occur between two other events e.g. hourly or daily windows would benefit from using tumbling windows. Applications which need to reason about the last n number of events, where n is known when the application is developed, would be more suited to using sliding windows.

5.1.3 Window Membership Expressions

Related to the question of what mechanisms to implement is the question of how to specify windows. Window membership is determined by a window membership expression, but how that expression is specified is an implementation issue that must be tackled. Options for the implementation of this window membership expression depend on the programming language exposed to the developer and range from using different window objects in an object-orientated programming language like Java to using a window membership function in C. The second issue concerning window membership expressions is how powerful the expressions will be. A number of implementation decisions must be made about the level of expressiveness including whether the expression can access attributes of the event whose membership of a window it is deciding, whether the expression can access other members of the window and if the expression can access other members of the event stream to which the window belongs. These are just three examples of the decisions that must be made regarding how expressive the window membership expression is. There is a trade-off in implementing these expressions. This trade-off is between making the specification of the expression as simple and easy to program, understand, test and debug as possible and between allowing sufficient levels of complexity in the expressions in order to make it powerful enough to implement the range of window membership functions that might be required in an application.

5.1.4 Execution Policy Specification

Deciding how expressive the execution policies should be is an important implementation decision. Similarly to window membership expressions, there is a trade-off between having highly expressive execution policies and simpler ones which are easier to program, understand, test and debug. Simple execution policies may not be able to represent the full range of possible situations that may warrant the execution of a multi-event handler, however they may be more than sufficient for the application that will use the implementation of the AESOP architecture. On the other end of the scale we can implement execution policies with all the expressiveness available, and allow arbitrary code to be executed in the execution policy. There is a also a decision to be made about the access available in the execution policy to attributes of the events. The execution policy could for example access timestamps and other attributes of the events, but of course this too increases the complexity of the execution policy. A decision must also be made as to whether or not the execution policies should have access to other events in the event stream. Execution policies are executed for every incoming event on any event stream of the multi-event handler. Therefore small increases in the execution time of the execution policy will have a detrimental effect on the performance of the application if the rate of incoming events is high. Expensive operations such as accessing the attributes of other events in the event stream windows should therefore be the exception rather than the rule in execution policies.

5.1.5 Programming Language

When deciding what language to implement AESOP in, a number of issues must be considered. The choice of declarative, functional or procedural programming language is a very important one and significantly influences the implementation of AESOP. As shown in chapter 3 declarative languages are extremely popular for specifying event composition. There are a number of reasons for this. Many event and stream processing systems have their roots in databases and declarative languages such as SQL are commonplace in specifying queries. Automatic query optimisation is another factor which is important to a number of these systems and declarative languages are well suited to query optimisation. Research is also quite advanced on optimisation of query plans in relational databases and this knowledge translates somewhat to the world of stream processing. STREAM [8] for example explicitly states that exploiting well-understood relational semantics as one of its design goals.

The full range of programming language styles including declarative languages may be used as the implementation language for AESOP. In fact two different languages could be used, one for the specification of execution policies and one for the implementation of the handler code. It might be considered advantageous, for example, to use a declarative SQL style language with suitable expressiveness to specify execution policy code while using a functional or procedural language to implement the handler code. This approach might be beneficial but could also prove confusing and overly complicated for the application developer.

5.2 Instantiations

To date we have two separate instantiations of the AESOP architecture. The two instantiations have been implemented on different platforms with varying requirements from the applications which use them. The first C-AESOP is implemented using the C language on the Gumstix platform while the second J-AESOP is implemented in Java on the Android smart-phone platform. In this section we describe both instantiations. For each, we first give an overview of the host event system which is extended and then discuss the mapping of the AESOP abstractions into the target language. Both instantiations are available on the CD accompanying this thesis.

5.3 C-AESOP

C-AESOP is an instantiation of the AESOP architecture intended for implementing computer-augmented sports systems, i.e., applications which augment sports artefacts with sensors and computational devices. With reference to the taxonomy presented in [115] the intended applications have application scope of *training*, *refereeing* and *safety*, but not *sports entertainment*. The intended environment for C-AESOP applications is up to and including bounded outdoor environments, but not unbounded outdoor environments. The intended sports are those with more than one artefact and any number of participants.

5.3.1 C-AESOP Requirements

There are a number of application characteristics that C-AESOP must support in order to be useful for implementing the types of applications it is designed for.

Distributed Communication

The multiple augmented artefacts in the application must be augmented with sensors and computational devices. These separate augmented-artefacts must communicate in some fashion. This calls for distributed communication in C-AESOP.

Distributed Over a Relatively Small Area

The sports that C-AESOP is intended to support have a fixed area and in general the maximum area of the sport will be in the order of 100 m^2, there are very few sports that operate within a bounded area that exceed this area.

Resource-Constrained

The devices used to augment these sports, if they are to be attached to artefacts used in the sport, must in general be light weight and with low power usage. This implies that resource-constrained devices will be the prevalent devices on which these applications will be implemented.

Relatively Small Number of Event Subscribers and Publishers

The number of participants and artefacts used in a sport are relatively low, below the order of hundreds of producers and consumers. A large-scale event system may have thousands of event producers and consumers, and specific event systems are designed to be able to support such activity. C-AESOP does not require support for high scalability with respect to number of subscribers and publishers.

High Frequency of Events

Applications that require tracking and analysis of human movement may produce a large quantity of events. Accelerometers are commonly used to analyse human movement, they usually produce from 50 to 200 readings a second, and any fine grained analysis of human movement will become more accurate the more readings are used in the analysis.

Near Real-Time Performance Required

Refereeing, safety and training where the three application scopes which C-AESOP is intended to support. All of these application scopes require prompt responses and results, in order to referee the sport, ensure the safety of the participants or provide advice on how to improve the activity being trained. The only exception is *in-vitro* training, where the athlete is training in a lab environment. It might be sufficient for the application to return results at a later period in time in such an application.

5.3.2 STEAM

As discussed in section 3.2.2, STEAM [99] is a distributed event based system designed for mobile ad-hoc networks. It uses an implicit event model, which means that consumers do not register with a mediator but instead subscribe to an event type and receive events from the STEAM system. The implementation of STEAM used in C-AESOP is written in C for the Linux platform. It uses multicast communication to deliver event notifications to all interested parties. Library functions are provided by the STEAM system to manage the subscription and notification of events. The STEAM event system does not retransmit lost events or reorder out of sequence events.

5.3.3 C-AESOP Time Model

C-AESOP has a *best effort* approach to timing. The events are not timestamped by the event system nor is global time maintained across the various nodes. Incoming events are not synchronised or delayed in any manner. This choice of time model is suitable for a range of applications in which the performance of the system as a whole is more important than the synchronisation of the events. This could happen for example in a system which needs to respond quickly to sensor data which is arriving at a high rate and is not tolerant of any delays in the processing of the sensor data.

5.3.4 Programming Language

The C language was chosen as the implementation language for C-AESOP for a number of reasons. The Gumstix platform is reasonably resource-constrained and C is a very efficient language and so uses fewer resources than most other languages. C is also well suited to implementations on the Linux platform and the STEAM middleware used as the host event system is also written in the C language.

Listing 5.1: Example Main Function in C-AESOP

```
1  int main(){
2    setup_steam();
3    stream* output =  (stream*) create_swing_output_stream(EMERGENCY_SUBJECT);
4    fall_detection_meh =create_multi_event_handler(3,inputs,output,handler,
         execution_policy);
5    inputs[0] = (stream*) create_accel_vector_input_stream(10,ACCELEROMETER_HIP,
         fall_detection_meh);
6    inputs[1] = (stream*) create_accel_vector_input_stream(10,ACCELEROMETER_WRIST
         ,fall_detection_meh);
7    inputs[2] = (stream*) create_accel_vector_input_stream(10,ACCELEROMETER_SHOE,
         fall_detection_meh);

9    while(1)
10     sleep(1000);
11 }
```

5.3.5 Event Streams in C-AESOP

In section 4.3 we described the event stream abstraction in AESOP. In the C-AESOP implementation we have two types of event stream input streams and output streams. We distinguish between input and output streams because input streams receive events and maintain event stream windows which are accessed in multi-event handlers, while output streams send the event onwards in the host event system. Input streams must therefore subscribe to events and create and maintain event stream windows, while output event streams must publish events and notify the host event system of new events. In the STEAM middleware consumers of events register their interest by subscribing to event subjects. They then receive all subsequent events which match this event subject. In C-AESOP event streams are identified using globally unique identifiers, which correspond to the STEAM event subject. These identifiers allow us to bind the input and output streams to our event streams. Input and output streams are used exclusively by individual multi-event handlers. This is in contrast to the event streams in the abstract model which are shared among the entire system. Event streams correspond to streams of STEAM events in C-AESOP. Therefore multiple input streams can bind to

a single event stream, by way of the event subject and multiple output streams can also bind to a single event stream by way of an event subject.

The sliding window mechanism is implemented in C-AESOP and is defined over input streams as these are unique to individual multi-event handlers. Tumbling windows are not supported in C-AESOP. Window membership in C-AESOP is based on window length, sliding windows are created with a length parameter and once the length of the window exceeds this length older events are discarded. There is no support for more advanced window membership expressions in C-AESOP.

Listing 5.1 shows how we declare and initialise event streams in C-AESOP. On line 3 we create the output stream that the multi-event handler will send events. We need only specify the subject used for the output events. On lines 5, 6 and 7 input streams are created. Sliding windows are the only windowing mechanism provided by C-AESOP so the input stream automatically creates a sliding window to handle events in the input stream. The first parameter of the create function is the length of the sliding window, in this case all the windows are created with a size of 10. This segment also shows the creation function for the multi-event handler. Among the parameters to this function are the function pointers to the execution policy and the handler function.

The data contents of the stream events in C-AESOP are typed. This allows us to send data encapsulated in any arbitrary C struct or basic type as content in an event stream of the same type.

Listing 5.2: Example Multi-Event Handler in C-AESOP

```
1  void meh_handler(accel_vector* hip, accel_vector* wrist,accel_vector* shoe,
       emergency* result){

3    struct accel_vector_stream_entry *np;
4    struct accel_vector_listhead *head = ((accel_vector_stream*)inputs[0]) ->
         head;
5    for (np = head->tqh_first; np != NULL; np = np->entries.tqe_next){
6      if(magnitude(&(np->value),3) > 15.0){
7        t = time(NULL);
8        local = localtime(&t);
9        strcpy(result -> time ,asctime(local));    //assigns the current time to
             the emergency
10       return;
11     }
12   }
13   head = ((accel_vector_stream*)inputs[1]) -> head;
14   for (np = head->tqh_first; np != NULL; np = np->entries.tqe_next){
15     if(magnitude(&(np->value),3) > 15.0){
16       t = time(NULL);
17       local = localtime(&t);
```

```
18          strcpy(result -> time ,asctime(local));
19          return;
20        }
21      }
22      head = ((accel_vector_stream*)inputs[2]) -> head;
23      for (np = head->tqh_first; np != NULL; np = np->entries.tqe_next){
24        if(magnitude(&(np->value),3) > 15.0){
25          t = time(NULL);
26          local = localtime(&t);
27          strcpy(result -> time ,asctime(local));
28          return;
29        }
30      }
31  }
```

5.3.6 Multi-Event Handlers in C-AESOP

The creation function for multi-event handlers takes as parameters an array of input streams, an output stream, an execution policy and a function pointer. The function pointed to in the function pointer has as many parameters as input streams with an additional parameter which is used for the result. The types of the parameters to the function are pointers to the corresponding type in the input and output streams. This allows the developer to write the logic for what to do each time the multi-event handler is called using the types as if they were writing a traditional C function. The C-AESOP middleware then manages the arrival of events, and supplies values to the appropriate parameters to the function. Any results of the multi-event handler by convention is assigned to the last parameter. When the middleware has executed the function it checks for a return value. If any value has been assigned to the result it then sends this value out on the output event stream. This allows for the instances where no value is returned by the multi-event handler - and so no onwards event is sent - and provides a clean and easy programming interface to the application developer.

Listing 5.2 shows the multi-event handler for the fall detection example application implemented in C-AESOP. The algorithm for the application checks the ten latest accelerometer readings stored in the sliding windows to see if the magnitude of the vector exceeds a threshold that might indicate that the change in posture of the patient is due to a violent fall. If the fall is deemed to be violent an emergency event is raised. C-AESOP uses UNIX tail queue lists as the data structure in which the events in the sliding window are stored. The lists are exposed to the developer as members of the relevant stream structure and they can traverse the list and access the events using standard UNIX mechanisms for accessing tail queue lists documented in the UNIX man pages [35]. This allows the developer to access all the events in the sliding window and access any attributes of those events.

Listing 5.3: Example Execution Policy in C-AESOP

```
1 //checks the hip accelerometer output to see if the patient has changed posture
2 int execution_policy(){
3   struct accel_vector_stream_entry *latest;
4   struct accel_vector_stream_entry *previous;
5   float threshold = 1; //radian value for threshold for when we decide the
        person is lying down
6   accel_vector upright = {0,-9.8,0};
7   if(!inputs[0] -> status)
8     return 0;      //if the hip accelerometer has not got a new reading

10  struct accel_vector_listhead *head = ((accel_vector_stream*)inputs[0]) ->
        head;
11  latest = head->tqh_first ;
12  previous = latest ->entries.tqe_next;
13  if(previous == NULL)     //only one value, don't execute
14    return 0;

16  if(angle_between((latest -> value) , upright   , 3) > threshold  &&
17     angle_between((previous -> value) ,upright   , 3) < threshold ){
18    return 1;  //change in posture detected
19  }
20 }
```

5.3.7 Execution Policies in C-AESOP

In C-AESOP an execution policy is a function that is called by the system when new data arrives on a stream belonging to the multi-event handler. The function decides whether or not to call the multi-event handler based on the arrival of this event data on an input event stream. Execution policies in C-AESOP are written as normal C functions. They take no parameters and return true or false depending on the outcome of the expression. A pointer to this function is passed to the create method when creating the multi-event handler. Execution policies in C-AESOP can access the full event stream windows and can access event data within these windows. Execution policies in C-AESOP are therefore extremely powerful, they have access to the full expressiveness of C when specifying when to execute the multi-event handler. Execution policies can also access the complete event data available to the multi-event handler including historical event data and can access the attributes of these events. Execution policies in C-AESOP should be kept relatively short and efficient. Because C-AESOP is designed for applications with a high frequency of events the code in the execution policies will be executed a large number of times, equal to the sum of the number of events that arrive for the

particular multi-event handler. Therefore having large or poorly written execution policies may have a very negative effect on the performance of the application.

Listing 5.3 shows the execution policy for the fall detection application implemented in C-AESOP. The execution policy checks the reading for the hip accelerometer. If the reading shows that the patient is reclining and the previous reading from the hip accelerometer shows the patient in an upright position, it is deduced that the patent's posture has changed from vertical to horizontal. The multi-event handler is called to determine if this change in posture was violent. The execution policy is written carefully to ensure that its execution will be efficient. If the event which caused its execution is not from the hip accelerometer it immediately returns, as this can not cause the execution of the multi-event handler.

5.4 J-AESOP

The second of our two instantiations of the AESOP architecture is called J-AESOP. It is implemented in Java on the Android platform. Android is a mobile smart-phone platform which has support for a rich array of sensors. J-AESOP was implemented on top of a custom event system based on the mediator event model [101].

J-AESOP is an implementation of AESOP that is intended for developing sensor-driven applications on a smartphone platform, in particular using the sensors in the phone to drive the behaviour of the user interface. There is a rapid rise in sensors on mobile devices and J-AESOP is intended to simplify the process of developing applications that take advantage of these features.

5.4.1 J-AESOP Requirements

Centralised Event System

In smartphone platforms sensors are integrated in the device. Therefore there is no need to have a distributed event system, a centralised event system is sufficient.

Handle High Frequency Events

The sensors in smartphones are types of sensors that produce a high rate of sensor readings. Accelerometers and magnetometers update at a relatively high rate, in the order of hundreds of readings per second. This high rate of events must be handled by J-AESOP.

Time Model

J-AESOP is intended to drive user-interfaces using the sensors on smartphones. The timing of these events is important for determining the actions that should be taken in the user interface. As the event model is centralised timestamps can be implemented easier than in a distributed model.

Execution Policy Expressiveness

J-AESOP is intended to be used to allow multiple sensors to drive the user-interface of a phone. The combination of multiple events should be as powerful as possible

Rapid Prototyping of Multi-Event Handlers J-AESOP is designed to support emerging applications that use sensors in smartphones. The entire application space is quite new and application developers are discovering how to program with sensors and create interesting applications using them. Therefore it should be easy and fast to explore new ideas using the sensors. Therefore there should be a minimum of programming effort required in specifying and testing new multi-event handlers.

5.4.2 J-AESOP Event Model

The host event system for J-AESOP is a custom event system developed by the author which follows the observer design pattern [46]. The event system uses a mediator event model [101] and is similar to the standard way event systems are constructed in Java, e.g., in JavaBeans [103]. A custom event system was developed for a number of reasons. The Android sensor manager implements an event like interface however this was not extended with AESOP. The SensorManager class in Android does not provide an interface for application developers to raise events, and so does not meet AESOP's requirements for a host event system. The Android source tree was under development and subject to change. Modifying the Android source code to allow application developers to notify events was considered unwise because of the difficulty in maintaining the code and the fact that any user of the subsequent application would need to install the modified operating system source code, which was not an option. Using a pre-existing event system was not possible when development was initiated on the application. At the time development began, in September 2008, there was only one event system which claimed to support Android [74], and their licensing fee was prohibitive.

In the custom event system event sources create events and notify a mediator of the new event. Other parties, interested in receiving events register their interest with the mediator. The event model that J-AESOP extends is not distributed so the mediator directly calls the callback for all the registered event listeners. The process for interacting with the host event system is quite straight forward. The functionality of the event system is encapsulated in an EventManager class. There are several methods which can be called on this class. Only one instance of the EventManager class is permitted at any one time. The singleton design pattern is used to guarantee this and the developer calls the static method EventManager.getInstance() to receive the reference to the event manager. The method registerEventListener() is used by event consumers to subscribe to a particular event type and unregisterEventListener() is used by consumers to unsubscribe from an event type. Lastly eventOccured(Event m) is used by event producers to communicate to the event manager that an event has occurred. The EventManager instance then checks the list of all registered listeners and notifies the consumers who are interested in an event of this type.

5.4.3 The J-AESOP Time Model

In J-AESOP each event is timestamped on creation. When an event is created in a multi-event handler the oldest timestamp from the incoming events which caused the execution of the handler is applied to the event as its timestamp. When composing multiple multi-event handlers we do not want to combine the results of multi-event handlers with events that are younger than they are. This situation can occur because movement of the event through a sequence of multi-event handlers can take a period of time, and if this time is greater than the time between two events in a stream which is to be combined at a later stage with the composite of that event, then we could end up combining an incoming newer event with a composite event which happened before it existed. This behaviour can be undesirable in an application. As a solution to this problem a multi-event handler can specify a *timing* stream. This is one of the multi-event handlers input event streams and is the stream with the youngest events in the multi-event handler. Events arriving on other event streams which are younger than the youngest event on the timing event stream are ignored until an event younger than them arrives on the timing stream.

5.4.4 Programming Language

The choice of Java as the language to implement J-AESOP in was constrained by the choice of platform. The Android platform has no support exposed to the application developer for other languages apart from Java/Dalvik. This limits the choices available for the implementation language of J-AESOP. Using the same language as the application programmer writes the core application in, is however advantageous as it reduces the training period for the programmer as they must just learn new abstractions as opposed to learning a new language and new abstractions. J-AESOP was designed for the Android platform, however, it will run on any platform which supports Java.

5.4.5 The AESOP to Java Mapping

J-AESOP is implemented in a manner which is true to Java's object-orientated nature. A base multi-event handler class is provided which must be extended to implement multi-event handlers, event streams and execution policies. When creating a new multi-event handler the developer extends the multi-event handler class, implements the handler method and the execution-policy method and instantiates the characteristics of the event stream class. In the following section we will discuss the AESOP to Java mapping in details.

Listing 5.4: The Multi-Event Handler class

```
1 package ie.ndrc.vgh.AESOP;

3 public abstract class MultiEventHandler {
```

```
5         public MultiEventHandler() {
6         }

8         public abstract boolean executionPolicy();
9         public abstract void handler();
10 }
```

5.4.6 The MultiEventHandler Class

The most important class from the point of view of the developer is the MultiEventHandler class shown in listing 5.4. This class encapsulates the entire multi-event handler. The base class has two abstract methods, handler() and executionPolicy(), which must be implemented by the application developer. The handler method has a void return type and the programmer can raise as many or as few output events in the handler using the underlying event model. The executionPolicy() method is called every time an event arrives in one of this multi-event handlers event streams. This execution policy decides if the handler should be called. Because the execution policy method is implemented in Java it has access to the full expressive power of Java. It also has access to all the attributes of the events in the windows of the event streams. This combination of a very expressive language and the information in the event streams allows very complex and powerful execution policies to be constructed.

Listing 5.5: Multi-Event Handler for Fall Detection Application in J-AESOP

```
1 public void handler() {
2     Iterator<Event> acceleration_1 = s1.getWindow().iterator();
3     Iterator<Event> acceleration_2 = s2.getWindow().iterator();
4     Iterator<Event> acceleration_3 = s3.getWindow().iterator();

6     while(acceleration_1.hasNext()){
7         if(vectorMagnitude(acceleration_1.next()) > 15){
8             call_ambulance();
9             return;
10        }
11    }
12    while(acceleration_2.hasNext()){
13        if(vectorMagnitude(acceleration_2.next()) > 15){
14            call_ambulance();
15            return;
16        }
17    }
```

```
18      while(acceleration_3.hasNext()){
19          if(vectorMagnitude(acceleration_3.next()) > 15){
20              call_ambulance();
21              return;
22          }
23      }



27  }
```

5.4.7 Example Multi-Event Handler

To give an example of the MultiEventHandler class in use we return to the example application, the patient fall detection application. The multi-event handler for this application is shown in listing 5.5. In the handler we retrieve the events in the window. These events are exposed using the iterator interface which is a common technique for accessing Java collections. We then iterate through the events calculating the magnitude of the acceleration vectors to determine if they exceed a threshold which in this example is hard coded to 15. If any of the acceleration vectors in the events exceed this threshold an ambulance is called.

Listing 5.6: Example Execution Policy from Fall Detection Application

```
2  public boolean executionPolicy() {
3      Iterator<Event> acceleration_1 = s1.getWindow().iterator();
4      MathVector upright_accel_vector = new MathVector(0,-9.8,0);
5      if (MathVector.angleBetween(acceleration_1.next(),upright_accel_vector) >
           threshold){
6          if(acceleration_1.hasNext() && MathVector.angleBetween(acceleration_1.
               next(),upright_accel_vector) < threshold){
7              return true; //posture has changed from vertical to horizontal
8          }
9      }
10     return false; // no change in posture

12 }
```

5.4.8 Execution Policy

Execution policies in J-AESOP are specified in the Multi-Event Handler class. When the application developer extends the MultiEventHandler class they must implement the abstract method executionPolicy(). This method takes no parameters and returns a Boolean to indicate whether the handler should be called or not. The execution policy can access the event windows over the streams by calling the standard event stream functions. Any Java syntax is permitted and any methods can be called from the execution policy, although it is advisable for performance reasons to keep code executed in execution policies reasonably fast as they get executed for every incoming event. Listing 5.6 shows the execution policy from the fall detection example application. In it the last two events are checked to see if there is a change in posture of the patient using the readings from the hip accelerometer. If one is detected the system calls the multi-event handler.

5.4.9 The EventStream Class

The EventStream class is the class that implements event streams in J-AESOP. Any event streams for the multi-event handler should be declared as instantiations of this class. The EventStream class supports length-based sliding windows, and can be accessed by calling the getWindow() method on an event stream class. This returns a List object which can be accessed using the standard Java iterator interface. The EventStream class must be instantiated with the type of the event in the event stream and the length of the window. The class then uses the underlying event model to subscribe to the relevant event. When a new event arrives, window membership is re-evaluated and the execution policy of the parent MultiEventHandler is invoked. If the execution policy returns true the handler for the parent MultiEventHandler is then executed.

J-AESOP supports length-based sliding windows. When an event stream is initialised one of the parameters passed to the method is the length of the sliding window. A list is then created of the corresponding length and when the list reaches the length specified the oldest event is discarded as every new event arrives, this maintains a length based sliding window over the stream. Each multi-event handler is self-contained and there is no sharing of event stream windows across multiple multi-event handlers. Window membership is determined on the basis of arrival with the last n events being members of the window, where n is the length of the sliding window.

5.4.10 Composition

J-AESOP allows multi-event handlers to be composed by allowing handlers to raise events using the underlying event model. Multi-Event handlers are allowed subscribe to these events in the exact same way as other simpler events which may originate outside the system. Care must be taken in the specification of the timing stream by the developer. Some algorithms are sensitive to situations where younger events are combined with composite events that are older than they are. This situation could arise if the length of time to create the composite event is larger than the arrival rate of a primitive event. To avoid this situation in J-AESOP the developer can specify a timing stream. The J-AESOP

system then guarantees that the youngest event in the timing stream will be the youngest event in the multi-event handler. All other events in the other event streams that are younger than this event are not delivered to the multi-event handler until a younger event arrives in the timing stream.

5.5 Summary

In this chapter we have discussed how to instantiate AESOP in a concrete architecture. We have discussed the major implementation issues that need to be addressed and the impact that these issues have on the resulting architecture. We have also presented two separate instantiations of the AESOP architecture C-AESOP and J-AESOP and showed how an example sensor-driven application would be implemented using both concrete architectures. The two implementations are quite different. These differences are caused by the different design decisions that each implementation must adhere to. Both implementations are very much influenced by the choice of host event system. C-AESOP is much less flexibile than J-AESOP in a number of dimensions and this is a result of the design pressure from being implemented on a resource constrained device as opposed to the relatively device rich Android.

This chapter documents the first part of the main contributions of this thesis. The implementations presented here are important as pieces of engineering in their own right and are usable to implement applications. However they are also useful as a guide to developers aiming to replicate the process of extending an event system with the AESOP model. The discussion of the implementation decisions that were made gives guidance to someone extending an event model and allows them to use our experiences extending two event models to their advantage.

Chapter 6

Evaluation

In this chapter we evaluate the AESOP architecture and its support for developing sensor driven ubiquitous computing applications.

Section 6.1 describes the strategy used to evaluate the AESOP abstractions. Sections 6.2 and 6.3 describe the two applications used to evaluate the support provided by AESOP for sensor-driven applications. Section 6.4 quantifies the cost of extending an event system with AESOP by measuring the overhead of the two AESOP instantiations described in chapter 5. Section 6.5 analyses the two applications presented in sections 6.2 and 6.3 and evaluates the functional support provided by AESOP for implementing sensor-driven applications. Section 6.6 analyses the generality of the abstractions. Finally section 6.7 summarises the achievements of the chapter.

6.1 Evaluation Strategy

In chapter 2 we derived a set of requirements which applications must satisfy in order to be suitable for supporting sensor-driven applications. These requirements were summarised in table 2.1. The requirements divide into two sections: support for developing sensor-driven applications and more general middleware requirements. In chapter 2, three requirements for abstractions to support the functional requirements of sensor-driven applications were derived. These were support for combination of sensor data, support for continuous data, and support for the combination of sensor data from multiple sources. In order to evaluate AESOP we must show that it supports these three functions. In terms of generality, four key areas were highlighted for sensor-driven applications. These were language generality, event model generality, application generality and platform generality. In order to show that AESOP is suitable for use in developing sensor-driven applications we must show that it is a general solution in these four dimensions.

Evaluation of programming abstractions is no easy task. Since, in essence, an abstraction's goal is to simplify the process of developing an application, quantifying how much simpler it is to write an application using the abstraction is difficult to do in an objective manner. There are many variables to consider and human experience is so wide that it may be impossible to come to a consensus on

what constitutes a good abstraction for a particular task. In fact it is not until after an abstraction has gained widespread adoption that we can say that it is a good abstraction, and even then there are bound to be dissenting voices. With a sufficient number of programmers it may be possible to implement the same application numerous times, some using the abstractions to be tested and some not, and measure some metric of the program, e.g., development time, number of bugs, lines of code or executable size. This exercise would be prohibitively expensive and the results would still be heavily influenced by the prior experiences and abilities of the programmers who implemented the applications. Factors such as programming skill, prior experience with similar abstractions or with similar applications would have a serious effect on the results of the experiment. These factors are also extremely difficult to control against. Also none of these metrics are particularly satisfying in determining how good an abstraction is. Therefore, it is both very difficult to reach a consensus on what a good abstraction is and it is also extremely difficult to measure objectively how good an abstraction is.

There are two main parts to the evaluation of AESOP. Firstly we analyse the functional support provided by AESOP for developing sensor-driven applications. We analyse how the AESOP abstractions can be used when developing applications to support the common characteristics of sensor-driven applications as discussed in section 2.3.1. In order to perform this analysis we need to analyse the AESOP abstractions in use in applications. Therefore, we present two case-studies of applications implemented using instantiations of AESOP. The level of functional support provided by AESOP for sensor-driven applications is evaluated by analysing the design and implementation of these two applications. The second part of the evaluation is the evaluation of the generality of the abstractions. We evaluate the generality of AESOP under the four dimensions presented in section 2.3.2. These are application generality, event model generality, programming language generality and platform generality. The two instantiations of AESOP presented in chapter 5 provide the basis for our evaluation of event model, programming language and platform generality and the two case-study applications are used in the evaluation of the application generality of AESOP.

In order to evaluate the AESOP abstractions we must use the abstractions in an application. To this end we have implemented two applications using the abstractions. These applications stemmed from two separate research projects in Trinity College Dublin. The first came from the SISTER project [55] which aimed to develop middleware for sensor-augmented sports. The application is a sensor-augmented squash racket which aims to aid the training of a squash player. The second application is from the Viking Ghost Hunt project [108], a collaboration between Trinity College Dublin and the National Digital Research Centre. It is a location aware game developed on a smartphone which aims to present a novel location based narrative to the user. The game takes advantage of the many sensors available in a modern smartphone and uses these to create a series of novel interface techniques using the handset.

The applications were chosen for this evaluation because they were part of ongoing research projects which the author was involved with. Both applications exist independently of the project one is a research project and the other is used in a location-aware game that is commercially released. We

can be reasonably confident therefore that they address users needs. In section 2.2 we derived the characteristics of sensor-driven applications. The two applications used in the evaluation posses these characteristics. Both of the applications have multiple sensors which they need to combine. They also both analyse historical sensor data and deal with asynchronous events of interest in the real world. The applications are therefore representative examples of sensor-driven applications.

The applications are presented as two case studies. We use these case studies in two separate parts of the evaluation. They are used initially when analysing the functional support the AESOP abstractions provide for developing sensor-driven applications. In this section of the evaluation we analyse the usage of the AESOP abstractions in the development of the applications and discuss the benefits gained from using the abstractions. The case studies are used the second time when evaluating the application generality of the abstractions. The implementations of AESOP, C-AESOP and J-AESOP as described in chapter 5 are each used to implement one of the applications. We also analyse the instantiations when we evaluate the generality of the abstractions with respect to event model, programming language and platform.

6.2 Squash Training Application

The use of technology in sport has a range of applications, including training, refereeing and injury prevention [115]. While the tradition of using technology for sport is long-standing (e.g., electric scoring systems for fencing have been in use since the 1930's [36]), recent advances in mobile computing and sensor technologies have given rise to a considerable range of sports systems that use the new technologies in interesting ways. Examples include indoor golf simulators [68], refereeing and entertainment systems for cricket and tennis [90], interactive climbing walls [88], sensor-augmented Taekwondo [34], and golf swing analysers [71, 117, 111, 39, 127]. The domain of technology-augmented sport systems has therefore increased considerably in complexity, not only with regards to the form of the solutions but also in respect to their functions and scope.

In the following section we will briefly describe the sport of Squash and discuss an application that was developed in Trinity College Dublin which aims to help squash players improve their technique. The application was developed using the C-AESOP instantiation of AESOP and we present an overview of the system design and its implementation.

The aim of this section of the evaluation is to demonstrate how AESOP supports sensor-driven applications and meets the requirements set out in chapter 2. In this section we will be demonstrating functional support for sensor-driven applications and go some way towards demonstrating the generality of the abstractions.

6.2.1 Squash

Squash is racket sport played in an indoor court between two competing athletes. Each athlete takes turns hitting a ball against the front wall of the court until either the ball is hit out of bounds or either athlete fails to hit the ball before it bounces for a second time. The sport is characterised by

extremely fast movement of both players and ball. In general the shots available to the player are either "drives", where the ball is hit off the front wall with the hope that it will bounce towards the back of the court in one of the corners, the "drop", where the ball is played to bounce in one of the front corners, or the "boast" where the ball is played from the back of the court into a side wall so that it will hit the front wall on the opposite side and bounce in the front corner.

6.2.2 Sensor-Augmented Squash Training

The application presented here is a training application which aids the squash player as they perform a drill using their squash racket. The shot which the application is aimed at is the "drive" shot. The aim of the application is to determine statistics that are interesting to the athlete about their stroke and provide useful feedback based on this information. The angle of contact between the squash ball and the racket head when performing a drive has a very strong influence on the quality of the resulting shot. It determines the spin of the ball which influences how quickly it "dies" after it bounces. If the racket is tilted towards the ceiling as it connects with the ball, the ball will tend to bounce lower and hit the ground faster on its second bounce than if the racket was facing straight at the front wall or towards the floor. Making the ball bounce lower and have its second bounce faster is generally desirable as it forces the opponent to react quicker and gives them less time in which to act. Advanced squash players tend to hit the majority of their shots with downward spin, while beginner and improving players usually hit their shots with no spin or only slight downward spin. The aim of the application presented in this section is to measure the angle that the player's swing makes when it comes into contact with the ball and to provide this information to the player. It is expected that this feedback could help train a player to hit the ball at the correct angle, but also could remind the player if their old habits reappear in subsequent training sessions.

The second major part of the application is the analysis of the weight distribution of the athlete as they make contact with the ball. Squash is a very fast energetic game with very little time to pause during play. Dominance of the centre of the court is key to controlling and winning a game. Each player therefore plays a shot and immediately attempts to recover to the centre of the court. This shot and recovery process is so crucial that it is generally attempted as one fluid motion with the player striking the ball as they are pushing back with their leading foot. In general the athlete also intends to transfer as much momentum as possible from their motion into the ball, which reduces the amount of power that needs to be generated from the arm and therefore helps the athlete conceal the shot. All of this focus on the stroke means that the motion of the athlete and their momentum at the point of contact with the ball is extremely important for the squash player and for the quality of the subsequent shot. This application aims to analyse the weight distribution of the player as they play the shot. We then aim to categorise the weight distribution and advise the player based upon this categorisation.

Sensed Phenomena In the field of sports bio-mechanics when analysing the motion of an athlete the concept of a *key point* is sometimes used. A key point is a particular moment of significance in

Figure 6.1: Start of Squash Swing

a motion. Key points are used to segment the motion into discrete moments that can be analysed separately. For example someone interested in analysing golf swings might consider four key points: addressing the golf ball, end of back stroke, contact with the ball and final position. By analysing these key points and the characteristics of the swing at these points (e.g., club head angle, head speed, hip position) the overall swing can be characterised and analysed. In this squash training application we are interested in three key points of the athlete's swing which are similar to the golf key points. They are:

- beginning of swing
- contact with ball
- end of swing

Figure 6.1 shows a squash player preparing to strike the ball. In the photograph the racket is being swung down and forward with the racket head above the players wrist while the players weight has been transferred onto the leading leg, i.e., the leg nearest the ball. Figure 6.2 shows a squash player at the moment of contact with the ball. As we can see from the photograph the racket is angled towards the roof of the court as it makes contact with the ball. The players weight is mostly on the leading leg, but on contact he will push back towards the centre of the court. Figure 6.3 shows the squash player after completing the swing. We can see that the squash racket has completed its follow through and the players weight has been transferred back towards the centre of the court.

There are a number of phenomena we must sense in order to implement this application. Firstly we must be able to detect all of these key points, and secondly we must be able to detect the angle at which the racket connects with the ball at the moment of impact. In order to detect the orientation of the racket we use a number of sensors attached to the racket. These sensors include accelerometers and orientation sensors. In order to detect the stance and weight distribution of the athlete when

Figure 6.2: Ball Contact

they make contact with the ball, we use pressure sensors embedded in the athlete's shoes.

Form Factor The form factor of the racket augmentation is very important. The rackets used to play squash are usually constructed of light weight materials such as graphite and must conform to strict guidelines as to minimum and maximum dimensions. Squash players are very sensitive to the weight and balance of their racket. Any changes to the weight and balance of the racket are felt by the athlete and may influence their performance while performing the swing. Design decisions were made to reduce the weight of various components, e.g., by opting for a smaller battery in order to reduce the weight of the device. In addition to the weight the placement of the device on the racket is also significant. To reduce the amount of the device that was on the outside of the racket a racket was modified to allow the computational unit and the battery pack to be embedded in the handle, leaving only the orientation sensor on the outside of the racket. This reduces the potential for the device to get damaged and by placing inside the racket reduces the effect the weight of the device would have on unbalancing the racket in the players hand when swinging and when contact is made with the ball.

6.2.3 The Gumstix Platform

The system is designed not to rely on infrastructure in the environment. This is so that the system can be used in different squash courts by the user and so that the system does not require any modifications to the court in which it operates. To make this as seamless as possible it was decided that the computational module should be embedded in the squash racket. The platform needs sufficient processing capability to perform analysis while remaining small enough to fit on the racket. The platform chosen to run the system is the Gumstix platform.

Figure 6.3: End of Squash Swing

The Gumstix platform is a small Linux-based embedded computer platform. It has support for a number of additional modules such as sensors and wireless communication. The platform comes with cross compiler support using the GNU Compiler Collection (gcc). The Gumstix platform was chosen as the platform for our squash training application as the processor is powerful enough to perform analysis of sensor data yet the form factor of the device is suitable for augmenting a squash artifact.

The Gumstix chip used in this application is the connex 400xm-bt [70]. It contains a 400MHz XScale processor with 64MB of RAM and 16MB of flash storage. It supports on-board bluetooth and provides several methods of connecting with external devices including I2C and USB.

Operating System

The Gumstix platform comes with a cut down version of Linux which uses the 2.6.27 Linux kernel and higher. It provides a cross-compilation environment which can compile both the operating system and any applications which can then be transferred to the device. Support for Linux allows application to be developed using standard Linux development tools on a desktop machine. The applications need only be cross-compiled and run on the Gumstix platform when developing platform specific functionality such the sensor interface at testing time. This flexibility greatly decreases the development time of the application.

Form Factor

The main processing board in the Gumstix measures 20mm x 80mm. An additional sensor interface board is required which is roughly the same dimensions. Batteries and sensors are also required and these significantly increase the size of the overall unit. The battery is potentially the largest of these additional components and the type of battery, the length of time the unit is to run and the types

Figure 6.4: Prototype Augmented Squash Racket

and number of sensors it is to power are the main factors in determining how big the battery should be. A lithium ion battery was chosen for the unit, because of its high power to weight ratio.

I/O and Sensor Support

The Gumstix boards combined with breakout expansion boards support a large amount of input/output that can be used to access sensors. This includes I2C, USB and serial connections. In this application we use an MTx sensor manufactured by Xsens [21]. It is connected by serial link to the Gumstix module. The MTx module contains a 3-axis accelerometer, a 3-axis magnetometer and an orientation sensor.

Figure 6.4 shows a photograph of an early prototype of the augmented squash racket with the Gumstix module attached to the outside of the squash racket. The module attached to the squash racket contains a 910 mAh battery, a Gumstix processing board with an attached breakout board, an MTx sensor unit and a voltage regulator.

6.2.4 System Design

The system design of the squash training application is shown in figure 6.5. As we can see there are a number of different components in the system. We describe here the role of each multi-event handler, the execution policy that controls it and the windows that are active on the input to each handler.

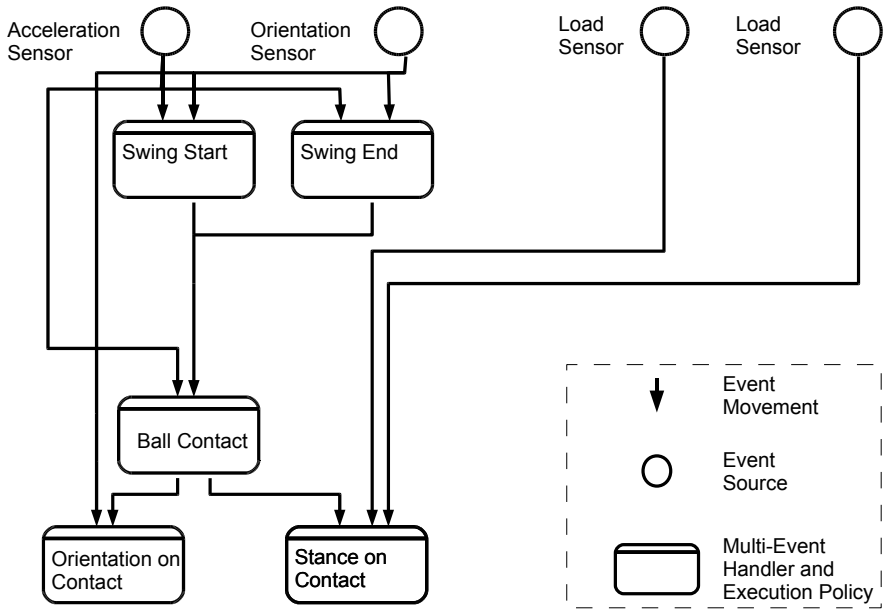

Figure 6.5: Squash Training Application

Swing Start Detection The *swing start* detection multi-event handler detects the start of a swing by the athlete. The handler combines the output from two sensors to detect this occurrence. In C-AESOP this is achieved by specifying two input event streams, the first for the accelerometer events and the second for the orientation events. The algorithm involves analysing the orientation of the racket and the subsequent changes to the acceleration to determine if a movement characteristic of the start of a squash swing is taking place. If such a movement occurs, a SWING event with content START is raised by the multi-event handler.

The execution policy for the *swing start* detection executes the multi-event handler whenever accelerometer data arrives on an input stream. This takes advantage of the fact that the orientation data will always be older than the accelerometer data. This makes sense from an implementation perspective as we are interested in the orientation of the racket before the swing has started.

End of Swing Detection The *end of swing* detection multi-event handler operates in a similar fashion to the *swing start* detection. The two multi-event handlers differ in that the *end of swing* detection is detecting the deceleration of the racket instead of the acceleration of the racket. The multi-event handler takes two input event streams, accelerometer events and orientation events. The execution policy executes the multi-event handler whenever acceleration events arrive on the input stream. The multi-event handler detects if the deceleration of the racket is similar to deceleration characteristic of the end of a swing and if the final orientation of the racket is similar to the end of a swing. If it is, a SWING event with content END is raised.

Ball Contact Detection of when a ball contact occurs is important as it is one of the key points at which we are interested in analysing the characteristics of the athlete's swing. We can detect contact

with a squash ball by examining the acceleration output of the accelerometer attached to the squash racket. A sudden reversal in the acceleration of the racket combined with a return to the previous values is characteristic of a ball contact. However any impact by the racket creates a characteristic spike in the accelerometer readings for example if the athlete were to drop the racket. We can reduce the amount of false positives we detect and reduce the computational effort required by only analysing the accelerometer data when we know a swing is in progress. This helps us distinguish between real ball impacts and other sudden changes in acceleration of the racket.

We can see from figure 6.5 that the *ball contact* multi-event handler takes two input streams, the swing event stream and the accelerometer event stream. In order to detect the ball contact correctly the multi-event handler must analyse the three latest historical sensor readings from the accelerometer and so the sliding window on the ACCELERATION event stream has length three. The execution policy executes the multi-event handler whenever an accelerometer event arrives and the latest SWING event indicates that a swing is in progress. If a ball contact is detected a BALL_CONTACT event is raised by the multi-event handler.

Orientation on Impact One of the main aims of the squash training application is to determine the orientation of the squash racket at the moment when it makes contact with the ball. The *ball contact* multi-event handler detects the moment that the racket comes in contact with the ball. This multi-event handler raises an event to signify this event occurring. The *orientation on impact* multi-event handler reacts to this event and to using the output of the orientation sensor determines the orientation of the racket at this moment. The orientation of the racket is measured in quaternions and the ball contact event contains a quaternion that represents this orientation.

The multi-event handler takes two input event streams, the BALL_CONTACT event stream and the ORIENTATION event stream. The execution policy for the multi-event handler states that when a BALL_CONTACT event occurs the handler should be executed. The multi-event handler takes the orientation input from the event stream and copies it to the event representing the orientation on impact. This event is then outputted in an ORIENTATION_ON_IMPACT event.

Stance on Impact The *stance on impact* multi-event handler implements the final stage of the second aim of this application which was to analyse the stance of the athlete as they connected with the ball. This multi-event handler takes three input event streams, the ball contact event and two events streams from the two load sensors in the athlete's shoes. The execution policy for the event states that when a ball contact event occurs, the handler should be executed. The multi-event handler needs to analyse the historical sensor readings from the load sensors in order to determine the stance of the athlete as they take the shot and so sliding windows of ten readings are used across both load sensor event streams. The multi-event handler analyses this historical data and raises a STANCE_ON_IMPACT event with a measure of the quality of the stance as its content.

Due to time constraints the functionality in the *stance on impact* multi-event handler was never tested with real-world data. The load sensors were not integrated with the Gumstix hardware so real

world load data of the athlete was not available to the application. However the modular nature of the multi-event handlers did allow us to test with simulated load sensor data.

6.3 Viking Ghost Hunt

The second case study we present is in the mobile gaming domain. The application is implemented on a smartphone platform. AESOP is used to combine events from the various sensors in the phone and to combine these events at higher levels to support the game logic. The game was developed for the Android platform and so uses the Java language. The implementation of AESOP used is J-AESOP which is described in section 5.4.

In the last section we demonstrated the functional support provided by AESOP for sensor-driven applications, in this section we also demonstrate this support. This section also provides additional evidence of the middleware generality of AESOP in terms of platform, event model, language and application.

6.3.1 Location-Aware Games

With the advent of Global Positioning Systems (GPS) and other forms of location sensing (e.g., wifi triangulation and mobile cell tower detection) and the prevalence of mobile computing platforms, games which adapt to the physical location of the player have become possible, if not exactly mainstream. Games such as *Pirates* [45], *Treasure* [30] and *Can You See Me Now?* [18] are examples of games which use location combined with a mobile device. As devices mature and offer additional features such as motion sensors, cameras and enhanced displays the user experience can become more complex while the complexity of implementing the games also increases. Of the location-based services listed above, GPS provides global coverage with the highest accuracy outside of small scale localised wifi triangulation. In the past GPS devices have not been widespread and have been used primarily in stand alone navigation devices. This has severely limited the amount of potential players for the location-aware games mentioned above. With the release of advanced smartphones such as the iPhone [65], HTC G1 [38], and Nokia N96 [109], GPS is now available to a wide range of consumers already integrated in a device capable of supporting a location-aware game.

6.3.2 The Game

Viking Ghost Hunt is a location-aware game set in Dublin city. The central premise of the game is that the player's phone is transformed into a paranormal detection device, which can be used to interact with ghosts from Dublin's Viking era. In the game, the motion of the phone is very important for a number of purposes. Augmented reality using images overlaid on the phone's camera feed is used to display ghostly imagery to the player, however knowledge of the orientation and motion of the phone is crucial to the illusion of the spectral scene. If the user was pointing the phone's camera at the ground and the ghost appeared upright, this would destroy the immersion in the game, similarly if

the user was moving the phone and the ghost appeared stationery, this too might destroy the player's immersion in the game.

6.3.3 The Android Platform

Programming Language

The Android platform [7] uses a modified version of Java called Dalvik. The language used is the same as the standard Java language however it compiles to Dalvik bytecode which is specific to the Android platform. The libraries supplied with the Android platform are different to the standard Java libraries, however the process of developing code is very similar to standard Java development and a plug-in is provided for Eclipse which simplifies the process of developing to and deploying the application to the phone. Several libraries are provided for Dalvik which allow the developer to access the hardware devices on the phone, most notably the accelerometers, magnetometers, camera and touchscreen.

Platform

The HTC G1 was the first phone to support Android. As the name suggests it is produced by the phone maker HTC. It has a 528 MHz processor, 192 MB ram, GPS, integrated 3-axis accelerometers and 3-axis magnetometers, touchscreen and qwerty keyboard. It was chosen as the development platform for the game as the open-source nature of Android was projected to significantly reduced the development effort and time and the open nature of the platform implied that the application would be easily modified to run on other Android phones that were yet to be released. The integrated 3-axis magnetometer was also seen as a major advantage over the iPhone 3G as directional audio was being investigated as an interface mechanism.

6.3.4 System Design

The Viking Ghost Hunt game interface is composed of several multi-event handlers. The overall system design is shown in figure 6.6. The multi-event handlers refine the raw sensor data into higher level sensors. The output from these sensors is then combined using more multi-event handlers. In this section we discuss the multi-event handlers in the system and the way in which the AESOP abstractions are used to simplify the development of the application.

Horizontal Detection The horizontal detection multi-event handler is designed to determine if the user is holding the handset horizontally in their hand in a manner that might suggest they were trying to look through it. As we can see from figure 6.6 this multi-event handler takes one input event, the handset's orientation. The execution policy executes the multi-event handler whenever a new event appears on the input stream. The algorithm for detecting if the handset is in a horizontal position checks the orientation of the handset to see if it is within certain thresholds that were determined experimentally. If the orientation of the handset is determined to be within these bounds

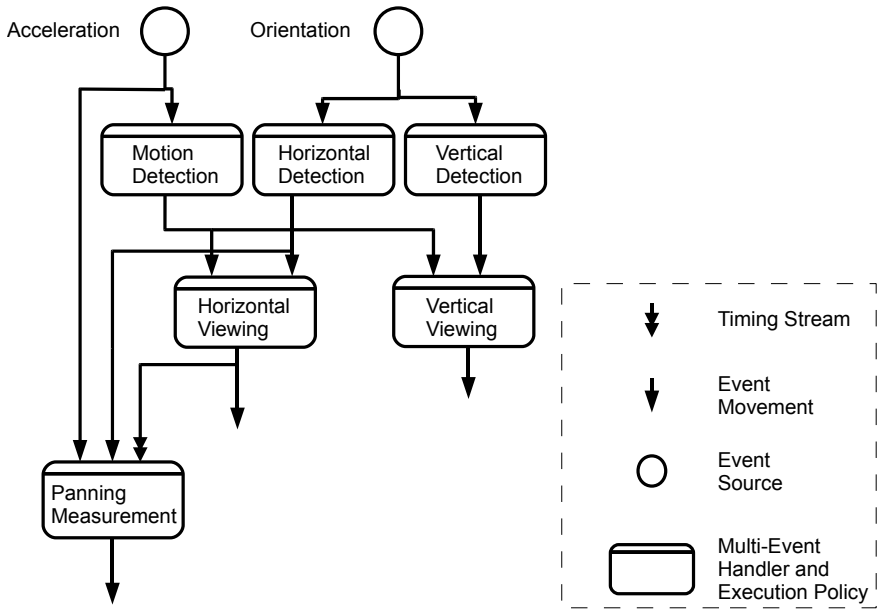

Figure 6.6: VGH system design

a horizontal event is raised. This multi-event handler raises events of type HORIZONTAL when horizontal orientation is detected.

Motion Detection The purpose of the motion detection multi-event handler is to determine how much movement of the handset is taking place. The multi-event handler's interaction in the overall system design is shown in figure 6.6. The algorithm used in the multi-event handler calculates the magnitude of the acceleration vector and checks it against pre-determined thresholds. This multi-event handler outputs NOT_STILL events when the handset is deemed to be moving. This multi-event handler is useful when implementing the user-interface, as determining if the handset is in motion is important for a lot of the user-interface functions. For example, if the handset is in motion above a certain threshold it is reasonable to assume that the user is not looking at the screen. Therefore important information, e.g., some animation advancing the plot of the game, should not be displayed at this time.

Horizontal Viewing Mode The purpose of horizontal viewing mode is to detect if the user is holding the handset in a manner that would suggest they are operating it as a camera. Horizontal viewing mode is used to drive *ghost-view mode* which is an augmented reality gameplay mechanism where the player can view ghostly happenings through their handset. Ghost view mode is shown in figure 6.7. Here we can see the user viewing a ghostly object, the spectral image of person being hung, through the camera. Several criteria must be satisfied to distinguish between situations where it is highly likely that the user is attempting to use ghost-view mode and situations where the user may be simply moving the handset. The first criteria that must be satisfied is that the orientation of the handset must be within certain bounds in the two axis that need to be restricted. This is performed

Figure 6.7: Viking Ghost Hunt: Ghost View Mode

by the horizontal detection multi-event handler and a HORIZONTAL event is raised when it occurs. The second criteria is that the handset must be stationary for a particular amount of time. This guards against situations where the handset briefly moves through a horizontal viewing position. The motion-detection multi-event handler outputs NOT_STILL events when the handset is not still.

In order to determine if the handset is in a horizontal viewing position for a specified amount of time the multi-event handler needs to combine the HORIZONTAL events and NOT_STILL events. Therefore the multi-event handler has two input event streams, one of type HORIZONTAL and one of type NOT_STILL. The multi-event handler does not need to refer to historical sensor data so the windows for the event streams are one event long in both cases. The execution policy for this multi-event handler specifies that when a new HORIZONTAL event arrives the multi-event handler should be executed.

As the events in J-AESOP are timestamped the handler needs only check the time of the last NOT_STILL event. This is automatically stored in the NOT_STILL event stream. If the event is older than a pre-specified amount, in this case two seconds, then the handset is determined to be in horizontal viewing mode and a HORIZONTAL_VIEWING_MODE event is raised.

We use the acceleration of the handset as an indicator of the handset's movement in this multi-event handler. It may be possible that the handset and user are accelerating at the same rate, e.g. if the user were in a car, and that the user may still be able to view the handset. However the Viking ghost hunt game is designed to be used by a player on foot and it would be highly unlikely, and probably unsafe, for a user to be able to maintain an acceleration above the threshold while playing the game.

Horizontal Panning Detection and Measurement This multi-event handler is used to detect and quantify panning of the handset when the user is viewing the handset horizontally. The panning

of the handset is used to make ghostly imagery that may be displayed on the handset appear to move relative to the background in a realistic manner, i.e., if the player moves the handset the imagery does not stay fixed to the centre of the screen and instead moves in the opposite direction to the motion of the handset. For this to be implemented we need to determine first what direction the handset is moving in and secondly we need some measurement of the magnitude of the movement in that direction.

Horizontal panning is only relevant when the handset is being held horizontally, therefore we only wish to execute the handler when the handset is in a horizontal position and we use the output from the horizontal viewing multi-event handler to determine this. We use the last horizontal viewing mode event as a ground truth of a point where the handset was held in a still position without accelerating. We then analyse the accelerometer readings since this horizontal viewing event and determine the direction of acceleration and the magnitude of the horizontal component of the vector. We output a panning event containing the direction and magnitude of the acceleration.

In order to analyse the accelerometer readings since the last horizontal view mode event we must store sufficient readings in a sliding window. The window on the accelerometer readings in this case is fifteen events long, and this is sufficient in this case. The windows on the horizontal viewing and horizontal detection event streams are one reading long as we do not analyse past the most recent event. The execution policy for this multi-event handler states that when a horizontal viewing event arrives and an accelerometer event arrives we execute the handler.

While acceleration and velocity are obviously not the same thing and a more accurate system would determine the velocity of the handset relative to the objects around it, in practise for small movements we can use acceleration to determine what direction the handset is moving in and give an approximation for the amount of movement taking place.

Panning detection is used when the handset is in horizontal viewing mode, so this implies that the algorithm can rely on at least one event which can be timestamped and act as a ground truth. We then compare the subsequent accelerometer vectors along the horizontal access of the handset summing them and determine what the acceleration of the handset is and what direction it is in. This gives us a measure of the movement of the handset which should be sufficient for detecting and measuring panning for the user interface. When a panning motion is detected a HORIZONTAL_-PANNING event is raised which contains a value which corresponds to the direction and magnitude of the movement.

Other multi-event handlers A large amount of other multi-event handlers have been developed in the prototyping stage of this application but did not make it to the final product. These include a *vertical panning* multi-event handler analogous to the horizontal panning one, a multi-event handler to distinguish between a *yes* gesture and a *no* gesture and a multi-event handler to recognise a *cast-out* gesture performed by the player.

6.4 Performance

In this section we analyse the performance of the two instantiations of AESOP. We measure the resource usage of the host event system and of the AESOP instantiations, and quantify the cost in terms of runtime resources of extending the two host event systems with AESOP. The purpose of this analysis is to quantify the overhead incurred as a result of extending an event system with AESOP. To measure the performance of each instantiation of AESOP we use the same application, a simplified version of the fall detection application from section 2.1.2. We use the same application so that there is consistency in our approach and because it allows us to simplify the experiment so that we can focus the on the performance of AESOP as opposed to the performance of the application.

This section is used to evaluate the platform generality of AESOP. We analyse the resource usage of AESOP and provide evidence to support the fact that AESOP has good platform generalit

6.4.1 C-AESOP Resource Usage

We analysed the resource usage of C-AESOP on the Gumstix platform. The resources that we measured are CPU utilisation, RAM use and disk storage.

Experimental Setup In order to perform the experiments we required an application written using the AESOP abstractions. The application used to conduct the experiments was the fall detection application introduced in section 2.1.2 and discussed throughout Chapters 4 and 5. The full source code for the multi-event handler is available in appendix B.1. The execution policy for the multi-event handler is complete however the multi-event handler code itself was removed. This was to prevent measurement of application code, and to focus on measuring the resource usage of the C-AESOP abstractions. To simplify the experiment and to ensure that we are measuring the performance of C-AESOP we replaced the sensors in the application with simulated acceleration sensors. The simulated sensors output an acceleration event with an appropriate event subject and pause for a configurable period of time.

To provide a measurement of the underlying resource usage of the STEAM middleware we also measure a simple STEAM application. The STEAM application subscribes to the same accelerometer events as the fall detection application. It is an event sink i.e., it receives the events and calls an empty event handler. Shell scripts were also developed to facilitate the experiments and to increase the accuracy of the timing of the execution of the applications. The full source code for the fall detection application, simulated accelerometer sensors, event sink and shell scripts are available on the CD that accompanies this thesis as appendix C.

The Gumstix connex platform was used, which is the platform used in the squash application discussed in section 6.2.1. The system was compiled using subversion version 1445 of the Gumstix buildroot. The Linux kernel was a modified version of 2.6.21 with multicast enabled, compiled using the buildroot environment. The version of uClibc which was used was version 0.9.29.

Experimental Technique The frequency of the accelerometer events was varied and the fall detection application was run three times for each frequency. Shell scripts were developed to accurately measure the execution time of the experiments and to repeat the experiments a large number of times with different parameters. The length of each run needed to be timed as it frequently exceeded the five minutes that was allotted to it in the shell script particularly at higher event frequencies. The application was timed using the `time` command from the Linux shell. The shell scripts reduced human error when timing the length of each run. The entire process was repeated replacing the event sink application for the fall detection application.

The number of events sent by the simulated sensors was also recorded. This figure was divided by the time the application was running in order to get a correct measurement for the event frequency of the accelerometer sensor.

To analyse the CPU utilisation of the applications we analysed the output from the `/proc/PID/stat` file. For each of the processes associated with the application we added the time that the process was in user mode and kernel mode. We divided this figure by the real time the application was running as determined using the `time` command in the shell script. The result of this calculation allowed us to calculate the percentage of time that the application was running that it was using the CPU, i.e. the CPU utilisation. The frequency of the sensors was calculated by dividing the number of events sent by the time the application was running. To measure the memory usage we used the process information available through `/proc/PID/smaps`. Finally to measure the size of the application in the file system we used the `du -h` command.

CPU usage Figure 6.8 shows a graph of CPU utilisation versus event frequency for C-AESOP. The full results of the experiment are available in table A.1. The event frequency column shows the total event frequency for the application. The total CPU time is measured in jiffies in the `/proc/PID/stat` file. Jiffies on the Gumstix platform are 10ms in length, this figure was converted to seconds by multiplying by .01. As we can see from the graph CPU utilisation is directly proportional to event frequency.

Figure 6.8 also shows the results of the experiment using the event sink in place of the fall detection application. The complete results of the experiment are available in table A.2. Each run was repeated three times and the event frequency and time of run were measured in the same manner as with the fall detection application. Again the CPU utilisation is directly proportional to the event frequency.

As we can see from figure 6.8 STEAM processes slightly more events than C-AESOP for each increase in CPU utilisation. This is to be expected as C-AESOP uses STEAM as its host event system and all the processing that is performed by STEAM must also be repeated in C-AESOP.

Figure 6.9 shows the CPU utilisation per event per second. We can see from the graph that it costs approximately .01% more CPU utilisation to process an event using C-AESOP then it does using just STEAM. This overhead remains constant as the event frequency increases.

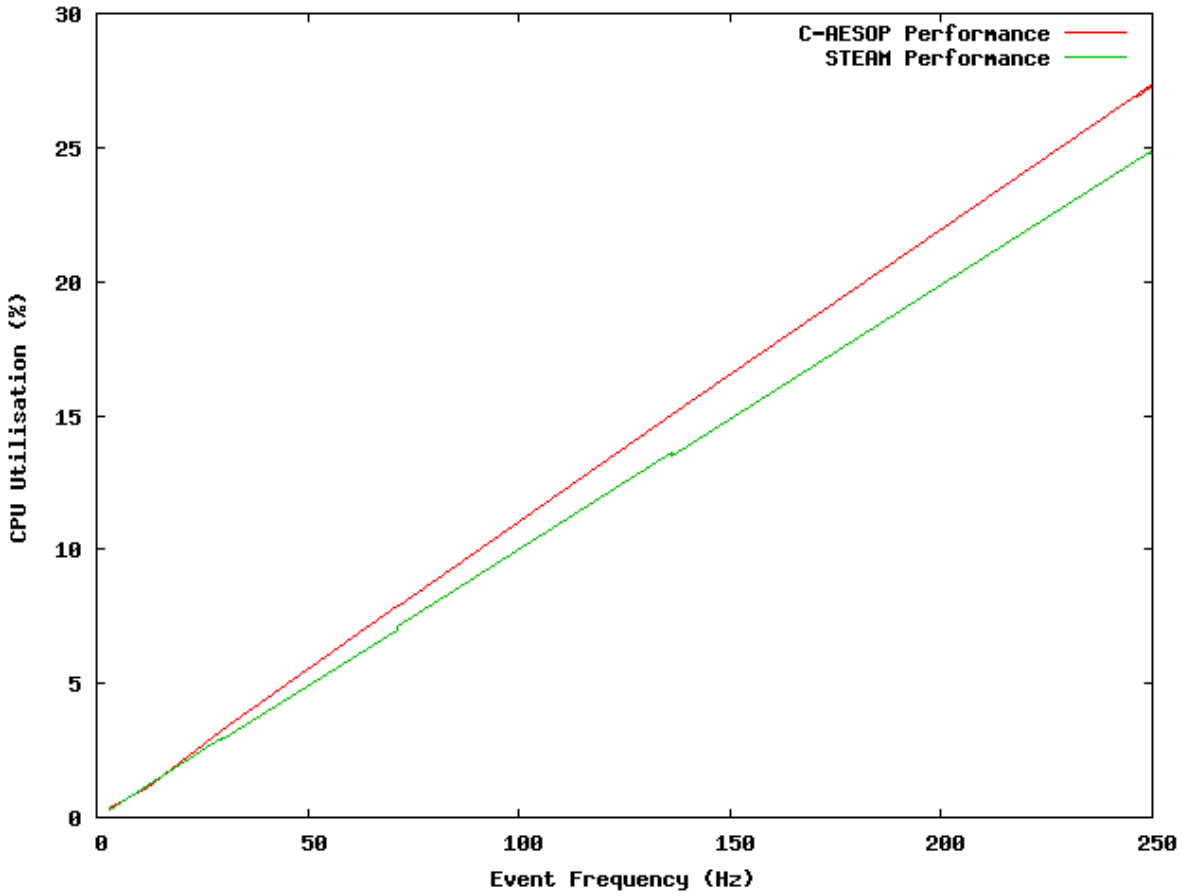

Figure 6.8: C-AESOP v STEAM Performance

	Total Available	STEAM	C-AESOP	Increase	Overhead %
RAM	109.5MB	124KB	144KB	20KB	16.13
Executable Size	14.8MB	34KB	31KB	3KB	9.68%

Table 6.1: STEAM and C-AESOP RAM Usage

RAM Usage The total memory of the process which is displayed using the top command also includes any shared libraries that the application may be using. We want to exclude these libraries from our calculations as we are interested in measuring the memory usage of the AESOP abstractions and C-AESOP is not implemented at this time as a shared library. To do this with some degree of accuracy we analysed the output from the result of issuing the `/proc/PROCESS_ID/smaps` for each of the processes attributed to the fall detection multi-event handler. The memory marked `Private_Dirty` and `Private_Clean` corresponds to memory that the process has used in memory, as opposed to memory which has been reserved and not used or that is shared with other processes. Table 6.1 summarises the RAM usage of STEAM and C-AESOP. All runs of the application were found to be using 144 KB of private memory. From analysis of the event sink application all runs of the application used 124 KB of private memory. Therefore the fall detection application uses 20 KB more than the STEAM only application. Within this 20KB is the program code of AESOP and also sliding windows of acceleration readings and code for the execution policies. Also, the page size

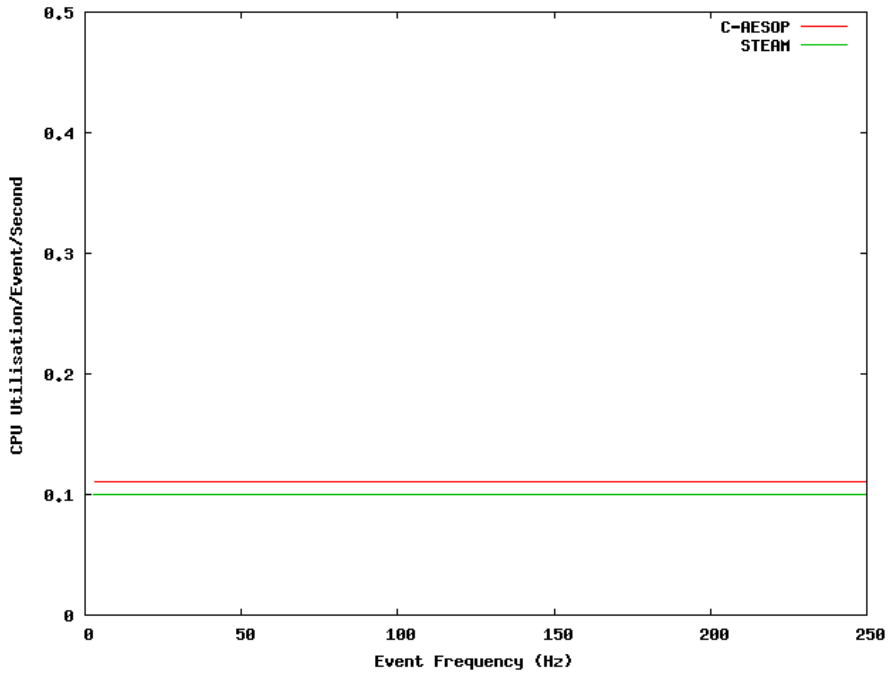

Figure 6.9: CPU Utilisation Relative to Event Frequency

is 4KB on the Gumstix platform so the resolution of this measurement is in 4KB increments, which restricts us from getting a more accurate measurement of the memory usage of the two applications. Memory profilers such as valgrind [43] are not available on the Gumstix platform.

Executable Size The size of the fall detection executable was measured at 34.0 KB. The STEAM event sink application was measured at 31.0 KB. The difference between the two applications is 3KB. This 3KB is the overhead for using the C-AESOP abstractions. The file system has 14.8MB total storage.

6.4.2 J-AESOP Resource Usage

The process of analysing the resource usage of the AESOP abstractions in J-AESOP on the Android smartphone proceeded in a similar manner to the analysis of C-AESOP on the Gumstix. We compare the Fall Detection application to an event sink application which consumes the same types and quantities of events.

Experimental Setup The Fall Detection example application introduced initially in section 2.1.2 was also used to analyse the performance of the J-AESOP abstractions. The initial fall detection application was implemented using the phone's accelerometer sensor. However to reduce the influence the phone's hardware might have on the performance of the application a dummy sensor was developed which outputted accelerometer sensor data at a configurable rate. This is so we can have a more accurate view of the resource usage of the abstractions as opposed to the hardware interface or the system calls that support them. Version 1.1 of the Android SDK libraries and Android Development

	Total Available	Custom Event System	J-AESOP	Increase	Overhead %
RAM	192MB	2.087MB	2.09MB	.003MB	.14%
Executable Size	256MB	28KB	32KB	4KB	14.28%

Table 6.2: J-AESOP Resource Usage

Tools (ADT) were used with version 3.4.2 of the Eclipse IDE to compile and deploy the application. Version 1.1 of the Android operating system was also used. The smartphone used to perform the tests was an Android developer phone version 1.1 [69].

Experimental Technique The Fall Detection application was deployed on the smartphone using the ADT. There is no easy way to measure CPU utilisation on the Android platform, e.g., the time utility used on the Gumstix platform to measure the CPU utilisation is not available in the shell. In order to get a rough metric for the amount of CPU used by the application, we used the debugging tools provided in Eclipse with the ADT plugin. The plugin allows us to periodically get an update of the amount of CPU time, in jiffies, that the individual threads of the process have been scheduled in user mode and in kernel mode. By summing together the user time and kernel time for each thread in the process we can calculate a figure for the amount of time the process has spent using the CPU. We measure the amount of time the application was running using log statements which are timestamped and output when the application starts and periodically throughout the execution of the application. By dividing the amount of time the application has spent using the CPU by the total time the application has been running we can calculate the CPU utilisation of the process. This figure provides us with a rough estimate of the CPU usage of the process which is sufficient for our use. In order to investigate the relationship between the frequency of the incoming events and the CPU usage of the application we varied the frequency of the incoming events. The amount of time the dummy sensor paused for is configurable and this was varied in five increments between a 250 millisecond delay and a 10 millisecond delay. The frequency of the sensor was calculated by recording the number of events and dividing this by the amount of time the application was running. The application was executed three times for each delay.

To measure the memory usage of the application the Eclipse ADT debugger was used. Heap updates were enabled on the application and the total allocated heap was read from the display.

The total size of the deployed application was determined by accessing the information on installed applications from the settings menu on the handset.

CPU usage Figure 6.10 shows the results of the experiments for both J-AESOP and the custom event system. The full data from both experiments are available in tables ?? and A.3. Figure 6.11 shows the increase in CPU utilisation for each additional event per second.

RAM footprint Table 6.2 summarises the RAM usage of J-AESOP. The size of the allocated heap was 2.09 MB for the fall detection application implemented using J-AESOP. When only the custom event system was used 2.087 MB of memory was allocated on the heap. Therefore the J-AESOP

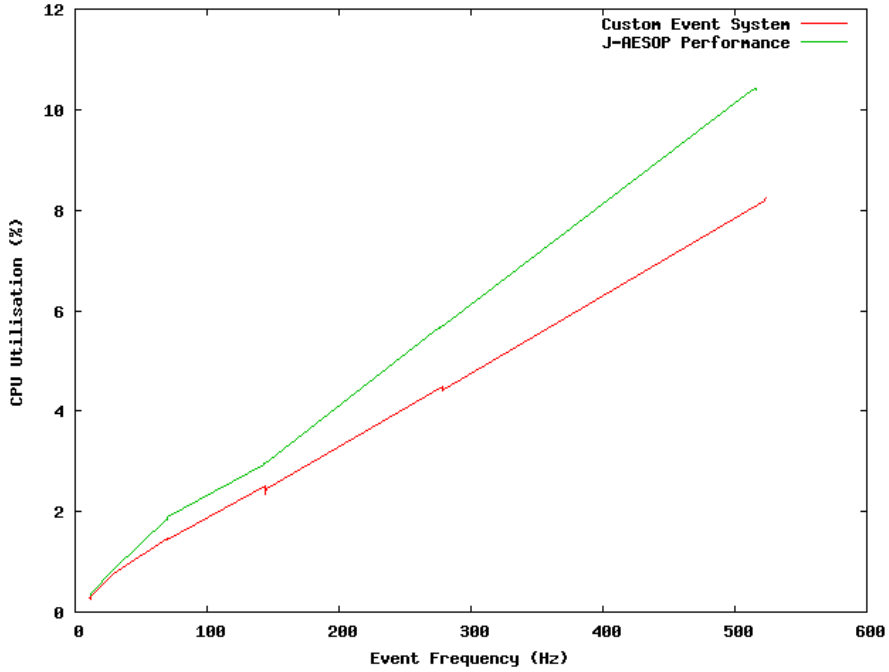

Figure 6.10: CPU Utilisation of Custom Event System and J-AESOP

abstractions consumed .003 MB of RAM. The total installed RAM available on the device is 192 MB.

Executable Size Table 6.2 also summarises the disk storage used by J-AESOP. The total size of the installed fall detection application was 32.0 KB. The application which only used the custom event model was 28KB in total. Therefore the J-AESOP abstractions used 4KB of storage. The total available storage on the device excluding the SD expansion card is 256 MB.

6.5 Functional Support for Sensor-Driven Applications

In this section we analyse the applications presented in sections 6.2 and 6.3. We discuss the support provided by the AESOP abstractions for developing the applications.

6.5.1 Sensor-Augmented Squash Training

Swing Start and Swing End The *swing start* and *swing end* multi-event handlers detect the start and end of the swing respectively. They both combine orientation and acceleration sensor data to determine if a swing has just begun or ended. These two multi-event handlers are relatively simple however they still benefit from using the AESOP abstractions. If the AESOP abstractions were not being used in this application, the developer would need to implement the functionality that receives and combines the sensor data and develop custom functionality to periodically evaluate the state of the racket and its movement.

Ball Contact The *ball contact* multi-event handler detects the impact of the squash racket with the ball. It does this by analysing historical data from the accelerometer sensor. To reduce both

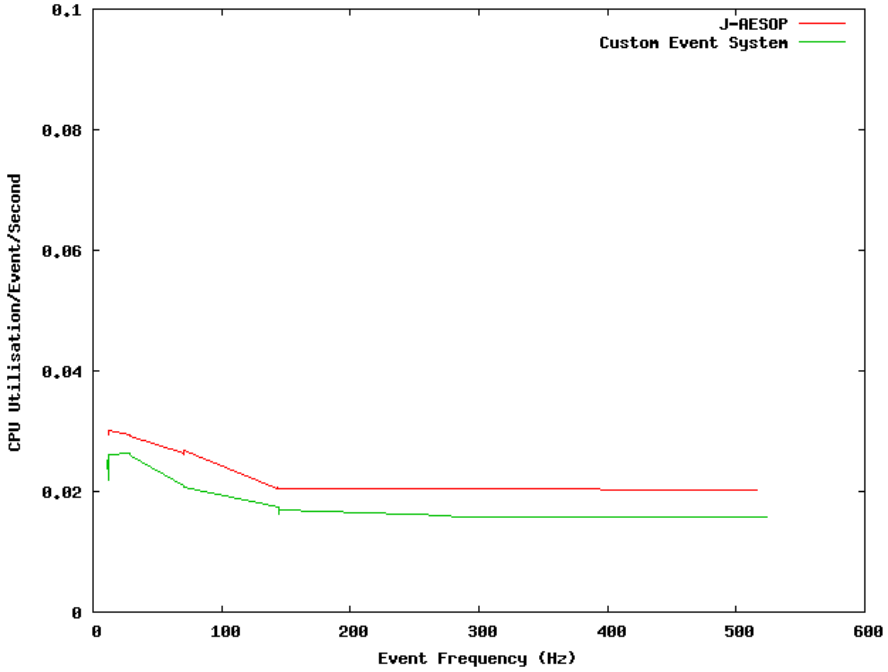

Figure 6.11: Relative CPU Utilisation of J-AESOP

the number of false positives and the amount of processing performed by the application, the multi-event handler only analyses the accelerometer data when a swing is in progress, as determined by the results of the *swing start* and *swing end* multi-event handlers. Implementation of this portion of the application benefits from using the AESOP abstractions. The execution policy allows us to separate the conditions under which we want to analyse the sensor data from the process of performing this analysis. This simplifies the design of the application. The second major benefit of the abstractions is in the use of the sliding window over the accelerometer event stream. The algorithm requires that the last three accelerometer sensor readings be analysed. Without the sliding window abstraction the developer would have to implement a very similar storage mechanism for historical event data. Use of AESOP in this case reduces the amount of time the developer would need to spend implementing and testing this feature. Using the event stream abstractions provided by AESOP also increases the readability of the code, once the reader understands the abstraction, and therefore decreases the cost of future maintenance. The process of combining the swing events and the accelerometer data is simplified and performed in a manner that is easy to understand.

Orientation on Impact The *orientation on impact* multi-event handler detects the orientation of the player's racket as they make contact with the squash ball. This multi-event handler combines two separate event streams. It combines the ball impact on the racket with the orientation of the racket at this point. If the AESOP abstractions were not being used the developer would have to write alternative C code to combine these two events. This would in effect recreate the function of this multi-event handler but in a more ad-hoc manner.

Stance on Impact The *stance on impact* multi-event handler analyses the weight distribution of the player as they make contact with the ball. It combines the data from the two load sensors with the output from the *ball contact* multi-event handler. It analyses the historical load sensor data to categorise the stance of the player as they perform their shot.

Without having access to the AESOP abstractions the functionality that this multi-event handler implements would be quite difficult to develop. There are a large amount of sensor streams that are combined to be able to analyse the stance of the athlete on impact with the ball, however when using the AESOP abstractions the multi-event handler code itself is relatively simple. The execution policy remains simple and the multi-event handler simply needs to specify the specific algorithm that is used to analyse the load sensor data. The storage of the historical load sensor data is also made significantly easier by the AESOP abstractions and is automatically available to the developer once they specify the event stream and the length of the sliding window.

6.5.2 Viking Ghost Hunt

Horizontal Detection,Vertical Detection and Motion Detection Both the *horizontal detection*, *vertical detection* and *motion detec*tion multi-event handlers are relatively simple multi-event handlers. They have simple execution policies, executing whenever there is an incoming event. The *horizontal detection* and *vertical detection* multi-event handlers analyse the orientation data raising the relevant events to represent horizontal or vertical orientation of the phone. The *motion detection* multi-event handler analyses the accelerometer events events whenever the smartphone is in motion.

The *motion detection* multi-event handler is also a relatively simple multi-event handler. It analyses the accelerometer data to

Horizontal Viewing Mode The *horizontal viewing mode* multi-event handler detects situations where the smartphone is being held in a manner that would suggest the user is operating it as a camera. The multi-event handler combines events from the *horizontal detection* and *motion detection* multi-event handlers in order to detect such a situation. Without the use of the AESOP abstractions the detection of *horizontal viewing mode* would be considerably more cumbersome. The two simpler multi-event handlers that detect if the handset is horizontal and if it is moving make the specification of this multi-event handler much easier. Both of these multi-event handlers could be incorporated into this multi-event handler, however the output from the motion detected multi-event handler is used elsewhere in the application. Using the AESOP abstractions in this instance reduces the amount of duplication in the application. This multi-event handler also demonstrates how the AESOP abstractions can be used to divide a problem into easy to solve pieces and compose these pieces to solve the original problem.

Horizontal Panning and Measurement The *horizontal panning and measurement* multi-event handler analyses the movement of the smartphone to determine the amount of horizontal panning that is taking place. Without the use of the AESOP abstractions the functionality in this multi-

event handler would be quite cumbersome to implement. This multi-event handler combines events from multiple higher-level sensors. It also analyses historical sensor data to determine the movement of the handset. Infrastructure to support these two elements of the functionality would need to be implemented by the developer. AESOP abstracts this functionality into a set of middleware components that the application developer can reuse in a structured and standard manner.

6.5.3 Discussion

AESOP simplifies the effort required to develop sensor-driven applications by providing abstractions to combine multiple sensor streams, analyse historical sensor data and perform both these functions at the same time by analysing multiple streams of historical sensor data. Composition of multi-event handlers is also a very powerful feature which allows application developers divide the functionality of the application into modules that are easier to implement and easier to re-use in other parts of the application. The AESOP abstractions allow the application developer to focus on the specific algorithms required to perform the analysis of the sensor data, rather than be concerned with developing the infrastructure required to combine or store the data.

6.6 Generality

Our description of the two applications shows that the AESOP model is useful for writing sensor-driven applications and supports the characteristics of those applications. However when designing middleware we are interested also in how general a solution the design is. In section 2.3.2 we discussed the notion of generality in middleware and discussed four dimensions of generality which are particularly relevant to middleware for sensor-driven applications. These dimensions are language generality, event model generality, application generality and platform generality. In this section we evaluate the generality of AESOP in these four dimensions.

6.6.1 Event Models

Section 2.3.2 discussed event model generality. Event model generality is particularly important for middleware which aims to be usable with a wide range of event models. Therefore it is important to show that the AESOP abstractions are usable with a wide range of event models.

In our discussion of the two instantiations of AESOP we have shown how the AESOP architecture maps to two very different event models. The STEAM event model is a distributed event system with an implicit event model, designed for ad-hoc networks using multi-cast delivery of events with no explicit time model. The J-AESOP host event model on the other hand is a localised event model which uses a mediator which handles event subscription and distribution and uses timestamped events. The AESOP model has been shown to successfully extend both of these event systems and the fact that the AESOP architecture makes few demands of the underlying event system ensures it will map to the majority of event systems. Therefore we conclude that the AESOP architecture has strong

support for event model generality.

6.6.2 Programming Languages

In section 2.3.2 we discussed programming language generality. As AESOP provides programming language abstractions it is important to know if these abstractions are implementable in the range of programming languages used for applications in the domain, or if it is limited to being used in a sub-set of the languages.

To show AESOP is general with respect to programming languages we must show that AESOP is mappable in the range of languages used in the domain. In a survey of frameworks for building ubiquitous computing applications [44], from 21 projects which specified support for a particular language or languages, 14 supported C/C++ and 12 supported Java. The next most popular language specified was Python with three projects supporting it, none exclusively. In embedded systems the C language is the most popular language, and its extension nesC [50], is by far the most popular language for writing wireless sensor network applications on the Mote platform. Java is also popular with resource constrained devices, e.g., on the Sun developed Sun Spot [105] and on the Android platform [7]. In this evaluation we have shown how the AESOP architecture maps to both the C and Java programming languages and as these two languages are used in the majority of ubiquitous computing applications, and sensor-driven applications are a sub-set of ubiquitous computing applications, we can conclude that the AESOP architecture is highly suitable as a general solution with respect to programming languages used for sensor-driven applications.

6.6.3 Applications

Section 2.3.2 discussed application generality and how it relates to middleware for sensor-driven applications. However, there are a large amount of possible applications which could be written and a large amount of application domains. Developing applications in all of the possible application domains would be prohibitively expensive and so doing an exhaustive evaluation of application generality is not possible.

In order to evaluate the generality of the AESOP abstractions with respect to application domain we return to the case studies presented in sections 6.2 and 6.3. The squash training application is a sensor-augmented sports training application. We have shown how the AESOP abstractions are useful for implementing this application and it can be reasonably assumed to be relevant to other similar applications in this domain. The Viking Ghost Hunt application is an example of an interactive media application on a mobile device. The AESOP abstractions are used to develop an innovative user interface which is driven by the smartphone's sensors. The AESOP abstractions have proved useful in this application also and it follows that other applications that use the phones sensors in a similar manner would be supported by AESOP. The two applications are very different sensor-driven applications and the AESOP abstractions are useful in both cases. While, two applications constitute less than exhaustive evidence we can be reasonably confident that the AESOP abstractions will be

useful for a wide range of sensor-driven applications.

6.6.4 Platform Generality

Section 2.3.2 discussed the notion of platform generality and why it was a desirable characteristic of middleware. In order to show that AESOP is a general solution with respect to platform we must show that the AESOP abstractions are suitable for use on the range of platforms which the middleware aims to support. In section 6.4 we analysed the performance of the two AESOP instantiations C-AESOP and J-AESOP. We compared the performance of the instantiations to the performance of the underlying host event system. By measuring the difference between the instantiations and their host event system we can quantify the overhead incurred by augmenting an event system with the AESOP abstractions. We compared the resource usage both to the resource usage of the host event system and to the available resources on the platform. This allowed us to quantify the amount of resources that the instantiation consumed and quantify how significant this amount is when compared to the available resources on the platform.

Figure 6.8 showed the CPU utilisation of C-AESOP compared to the CPU utilisation of the underlying STEAM event system. As we can see from the graph at a total event frequency of 250 Hz approximately 2.5% additional CPU utilisation is consumed. Figure 6.9 shows the relative difference in CPU utilisation between STEAM and C-AESOP. We can see from the graph that C-AESOP requires an additional .01% CPU utilisation per additional event per second. Figure 6.10 compared J-AESOP with its host event system. We can see from the graph that there is approximately a 2% increase in CPU utilisation for a total event frequency of 500 Hz. Figure 6.11 shows the increase in CPU utilisation of J-AESOP relative to the custom event system. We can see from the graph that the percentage overhead in CPU utilisation is less than .005% per event per second. We consider both this overhead and the C-AESOP overhead to be very modest increases in CPU utilisation.

When we consider the additional RAM consumed by both the C-AESOP and J-AESOP instantiations it is constant with respect to event frequency. C-AESOP consumes an additional 20Kb of RAM which is a 16.66% of the RAM used by the host event system, or less than .003% of the total available RAM on the platform. When we analyse the J-AESOP RAM usage it uses an additional .003 MB or less than .001% of the available RAM on the system. Both of these figures are tiny amounts of memory even on these resource-constrained platforms. The RAM usage also has the desirable characteristic that it remains constant despite increased event throughput.

Analysing the storage usage of the two instantiations, the C-AESOP instantiation uses an additional 3KB of storage which is 9.6% of the STEAM implementation or .2% of the total available storage. The J-AESOP instantiation uses an additional 4KB of storage which is 14.2% of the host event system or .0015% of the total available storage.

When compared to the resources available on the devices it is clear that the overheads incurred from using the AESOP abstractions are reasonable and a tiny fraction of the resources available on the platforms. We have analysed two instantiations of AESOP from the lower end of the devices on

which sensor-driven applications are developed. We would expect the results to hold across platforms with more resources than the platforms analysed. We have not analysed platforms at the extreme lower end of the resource spectrum, e.g. embedded micro-controllers and motes. However we would expect any platform that has the resources to support an event system, to be able to support the AESOP abstractions. On this basis we conclude that the AESOP abstractions are a suitable solution for platforms with similar or more resources than the Android and Gumstix platforms. Furthermore, we expect that any platform which supports an event system could be extended with an AESOP instantiation with similar overheads to the instantiations presented in this thesis.

6.7 Summary

In this chapter we have evaluated the AESOP model and its support for sensor-driven ubiquitous computing applications. We have evaluated the degree of support provided by AESOP for the requirements of sensor-driven applications as discussed in chapter 2. We have shown that AESOP meets the functional requirements of sensor-driven applications as described in section 2.3.1 by analysing two applications developed using two different instantiations of AESOP. The support for combination of sensor data in AESOP is demonstrated in both the squash and gaming applications. The usage of historical data is also demonstrated in both. The support for the analysis of historical sensor data is also demonstrated in the two applications. We can conclude therefore that AESOP provides abstractions to support historical sensor data, the combination of sensor data and the analysis of historical sensor-data from multiple sensors.

These applications are from different application domains demonstrating the application generality of AESOP. The instantiations used extend two very different event models which shows the event model generality of AESOP. Each application is written in a different programming language, Java and C respectively, which have been shown to be overwhelmingly the most used languages in the field of ubiquitous computing. This demonstrates the language generality of AESOP. Finally we have performed an in-depth analysis of the resource-usage of the AESOP abstraction. We have analysed the two instantiations running a cut down demonstrator application using simulated sensors in order to get consistent and comparable results. We have analysed the memory usage, the storage footprint and the CPU usage of the application for each instantiation. We have compared this to the underlying event model. Both instantiations of AESOP perform extremely well with a very low overhead and so we can we conclude that the AESOP abstractions are general with respect to platform.

Chapter 7

Conclusion

This chapter summarises the achievements of this thesis, places them in a greater context and outlines promising directions for future work.

7.1 Achievements

This thesis addressed the area of abstractions to support the development of sensor-driven applications. Motivation for this work came from the fact that event based programming is the predominant development technique for building ubiquitous computing applications however it is ill-equipped for meeting the requirements of sensor-driven applications. Stream processing systems have useful abstractions for building sensor-driven applications however they are not generally suitable for building ubiquitous computing applications. They also generally lack the ability to perform custom analysis of historical stream data from multiple sensors, which is a feature of the more advanced sensor-driven applications.

The thesis introduced the concept of sensor-driven applications to describe a subset of ubiquitous computing applications that modify their behaviour in response to readings from their sensors. Four example applications were presented and these were analysed and the characteristics of sensor-driven applications examined. Requirements for middleware to support sensor-driven applications were synthesised from these characteristics.

The thesis reviewed the state of the art in abstractions that support sensor-based applications. There were a number of different sub-categories in the state of the art review, reflecting the different approaches available for different systems. Two representative event-based models STEAM and JavaBeans were chosen for review and these were shown to have no support for the combination of sensor data from multiple sensors and no support for dealing with the continuous nature of sensor data. Two traditional event systems which had been extended with the abstraction of composite events, Cambridge Event Architecture and a generic extension to event based systems supporting composite events were also reviewed. These system were found to have more support for the combination of sensor streams but limited support for historical sensor streams. Two complex event processing systems,

SASE and Cayuga were also reviewed and shown to support combination of sensor streams and support for historical sensor data, however lacking the complex analysis that is required in sensor-driven applications. Two stream processing systems Aurora and STREAMS were also reviewed and shown to support a large number of the requirements of sensor-driven applications with limited support in Aurora for custom analysis of historical sensor data from multiple sensors. TinyDB and SINA, two sensor database implementations were also reviewed to give an additional insight into sensor-based applications and were shown to have some support for historical sensor data and multiple sensors, but to be unsuitable for sensor-driven applications.

The main contribution of this thesis is an extension for event based systems designed to add support for sensor-driven applications. This extension called AESOP, has a minimal set of requirements and would be suitable for extending the majority of event systems. AESOP has three main abstractions which extend event-based programming in order to support sensor-driven applications. These are event streams, multi-event handlers and execution policies. Two instantiations of the extended model, C-AESOP and J-AESOP were presented in chapter 5. The instantiations and the description of their implementation are also significant pieces of the contribution. They show how the abstractions can be implemented and the design pressures and trade-offs that need to be made when implementing the abstractions.

The two applications analysed in chapter 6 serve to verify the applicability and usability of AESOP in the design and implementation of sensor-driven applications. Specifically, we analysed the support provided by AESOP for the functional requirements of sensor-driven applications as outlined in chapter 2. By analysing the two instantiations of AESOP we showed that the model is general with respect to programming language and event model. By examining the two case study applications we showed that the AESOP model was a general solution with regards to application. Finally by measuring the resource usage of both instantiations of AESOP we calculated the overhead incurred by extending an event system with AESOP and showed that it was within reasonable bounds.

7.2 Perspective

Over the course of the five years that the author has been working with sensor-driven applications many significant changes have taken place. Cheap sensors such as accelerometers, magnetometers, RFIDs, light sensors and gyroscopes have become prevalent and made sensor-driven applications with a large number of sensors affordable. The Wii console revolutionised game interfaces by integrating accelerometers and cameras in the game controller. Meanwhile accelerometers have become commonplace in many consumer appliances including phones, laptops and media players. Shock detectors are used in laptop hard drives, and accelerometers are used in phones to automatically adjust the display based on the orientation of the phone. Additional sensors are also becoming commonplace including magnetometers, GPS, and gyroscopes (as seen by the extension to the Wii console with the Wii motion+). Sensors have also been installed in athlete's shoes and linked to media devices in the Nike+ effort between Nike and Apple.

The upward trend in the number of sensors incorporated into everyday objects is taking the ubiquitous computing vision closer to reality. However despite the large number of sensors available, applications which combine data from two or more sensors are still relatively rare. Hopefully the AESOP model will go some way towards simplifying the development process for sensor-driven applications and help make the ubiquitous computing vision a reality.

7.3 Future Work

With every piece of work brought to completion new questions arise and new challenges are identified. Timing schemes for composition of muti-event handlers and sharing streams in instantiations are two such issues.

7.3.1 Timing Schemes for Composing Multi-Event Handlers

In section 4.7 we discussed timing issues which can potentially arise when combining multi-event handlers. In section 5.4.10 we discussed one particular timing scheme which solves this problem in the J-AESOP model. However, there are a large amount of possible schemes that could be developed with different design considerations. The use of a timing scheme depends heavily on the time model of the host event system as well as the requirements of the applications. Further work is needed to develop and characterise these possible timing schemes.

7.3.2 Sharing of Event Streams

At present the two instantiations make no attempt to share the storage of event streams between multiple multi-event handlers on the same device which may be interested in events of the same type. Obviously for larger streams storing the same data multiple times is not efficient. The problem is not entirely straight-forward as different multi-event handlers can have a different view of the stream, i.e. what events have been consumed and what window over the event stream is meaningful to the multi-event handler. In fact if two multi-event handlers had radically different windows e.g. one sliding window of the last 100 events and one window based on an attribute of the event there may be little or no duplication of events in the windows of each multi-event handler. However, in a large amount of instances where an event stream is in use multiple times on the same node some sharing of the event stream would prove beneficial.

Bibliography

[1] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, J. h. Hwang, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. Zdonik. Aurora: A data stream management system. In *In ACM SIGMOD Conference*, page 666, 2003.

[2] Daniel J. Abadi, Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management, 2003.

[3] Asaf Adi and Opher Etzion. Amit - the situation manager. *The VLDB Journal*, 13(2):177–203, May 2004.

[4] BMW AG. http://www.bmw.com/com/en/insights/technology/technology_guide/start.html, Nov. 2009.

[5] Information Age. The main event. http://www.information-age.com/channels/development-and-integration/it-case-studies/441926/the-main-event.thtml, June 2008.

[6] Yanif Ahmad, Bradley Berg, Ugur Cetintemel, Mark Humphrey, Jeong-Hyon Hwang, Anjali Jhingran, Anurag Maskey, Olga Papaemmanouil, Alexander Rasin, Nesime Tatbul, Wenjuan Xing, Ying Xing, and Stan Zdonik. Distributed operation in the borealis stream processing engine. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 882–884, New York, NY, USA, 2005. ACM.

[7] Android. http://www.android.com/, Oct. 2009.

[8] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. Stream: The stanford data stream management system. Springer, 2004.

[9] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.

[10] Ken Arnold and James Gosling. *The Java programming language (2nd ed.)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.

[11] E. A. Ashcroft and W. W. Wadge. Lucid, a nonprocedural language with iteration. *Commun. ACM*, 20(7):519–526, 1977.

[12] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Dilys Thomas. Operator scheduling in data stream systems. *The VLDB Journal*, 13:2004, 2003.

[13] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16, New York, NY, USA, 2002. ACM Press.

[14] Brian Babcock, Mayur Datar, and Rajeev Motwani. Load shedding for aggregation queries over data streams. In *In ICDE*, pages 350–361, 2004.

[15] J. Bacon, J. Bates, R. Hayton, and K. Moody. Using events to build distributed applications. pages 148–155, Jun 1995.

[16] Jean Bacon, Ken Moody, John Bates, Richard Hayton, Chaoying Ma, Andrew McNeil, Oliver Seidel, and Mark Spiteri. Generic support for distributed applications. *Computer*, 33(3):68–76, 2000.

[17] Magdalena Balazinska, Hari Balakrishnan, Samuel R. Madden, and Michael Stonebraker. Fault-tolerance in the borealis distributed stream processing system. *ACM Trans. Database Syst.*, 33(1):1–44, 2008.

[18] Steve Benford, Andy Crabtree, Martin Flintham, Adam Drozd, Rob Anastasi, Mark Paxton, Nick Tandavanitj, Matt Adams, and Ju Row-Farr. Can you see me now? *ACM Trans. Comput.-Hum. Interact.*, 13(1):100–133, 2006.

[19] Sumeer Bhola, Robert Strom, Saurabh Bagchi, Yuanyuan Zhao, and Joshua Auerbach. Exactly-once delivery in a content-based publish-subscribe system. *Dependable Systems and Networks, International Conference on*, 0:7, 2002.

[20] Lars Brenna, Alan Demers, Johannes Gehrke, Mingsheng Hong, Joel Ossher, Biswanath Panda, Mirek Riedewald, Mohit Thatte, and Walker White. Cayuga: a high-performance event processing engine. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1100–1102, New York, NY, USA, 2007. ACM.

[21] Xsens Technologies B.V. 3d motion tracking - xsens. http://www.xsens.com/, Jan. 2010.

[22] A. T. Campbell, S. B. Eisenman, N. D. Lane, E. Miluzzo, R. A. Peterson, Hong Lu, Xiao Zheng, M. Musolesi, K. Fodor, and Gahng-Seop Ahn. The rise of people-centric sensing. *Internet Computing, IEEE*, 12(4):12–21, 2008.

[23] Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams: A new class of

data management applications. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 215–226. VLDB Endowment, 2002.

[24] Don Carney, Ugur Cetintemel, Alex Rasin, Stan Zdonik, Mitch Cherniack, and Mike Stonebraker. Operator scheduling in a data stream manager. In *In VLDB*, pages 838–849, 2003.

[25] J. B. Carter, D. Khandekar, and L. Lamb. Distributed Shared Memory: Where We Are and Where We Should Be Headed? In *Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, May 1995.

[26] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S. k. Kim. Composite events for active databases: Semantics, contexts, and detection. In *In Proc. of the VLDB Conference*, pages 606–617, 1994.

[27] S. Chakravarthy and S. Varkala. Dynamic programming environment for active rules. pages 3–16, 0-0 2006.

[28] Sharma Chakravarthy and Raman Adaikkalavan. Events and streams: harnessing and unleashing their synergy! In *DEBS '08: Proceedings of the second international conference on Distributed event-based systems*, pages 1–12, New York, NY, USA, 2008. ACM.

[29] Sharma Chakravarthy and Deepak Mishra. Snoop: An expressive event specification language for active databases, 1993.

[30] Matthew Chalmers, Marek Bell, Barry Brown, Malcolm Hall, Scott Sherwood, and Paul Tennent. Gaming on the edge: using seams in ubicomp games. In *ACE '05: Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*, pages 306–309, New York, NY, USA, 2005. ACM.

[31] Sirish Chandrasekaran, Sirish Ch, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Sam Madden, Vijayshankar Raman, Fred Reiss, and Mehul Shah. Telegraphcq: Continuous dataflow processing for an uncertan world, 2003.

[32] Jianjun Chen, David J. Dewitt, Feng Tian, and Yuan Wang. Niagaracq: A scalable continuous query system for internet databases. In *In SIGMOD*, pages 379–390, 2000.

[33] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. Niagaracq: a scalable continuous query system for internet databases. *SIGMOD Rec.*, 29(2):379–390, 2000.

[34] Ed Chi. Pervasive computing in sports technologies: Guest editor's introduction. *IEEE Pervasive Computing*, 4(3):22–25, 2005.

[35] Unix Community. Queue. http://linuxmanpages.com/man3/queue.3.php, Jan. 2010.

[36] Wikipedia Contributors. Fencing. http://en.wikipedia.org/wiki/Fencing, Mar. 2010.

[37] Owen Cooper, Anil Edakkunni, Michael J. Franklin, Wei Hong, Shawn R. Jeffery, Sailesh Krishnamurthy, Fredrick Reiss, Shariq Rizvi, and Eugene Wu. Hifi: a unified architecture for high fan-in systems. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 1357–1360. VLDB Endowment, 2004.

[38] HTC Corporation. http://www.htc.com/www/product/g1/overview.html, Oct. 2009.

[39] cSwing. cswing. Last accessed 14 October, 2010.

[40] Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta. The jedi event-based infrastructure and its application to the development of the opss wfms. *IEEE Transactions on Software Engineering*, 27(9):827–850, 2001.

[41] A. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. White. Cayuga: A general purpose event monitoring system. In *Proc. CIDR*, pages 412–422, 2007.

[42] Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. Towards expressive publish/subscribe systems. In *In Proc. EDBT*, pages 627–644, 2006.

[43] Valgrind Developers. Valgrind. http://valgrind.org/, Feb. 2010.

[44] Christoph Endres, Andreas Butz, and Asa MacWilliams. A survey of software infrastructures and frameworks for ubiquitous computing. *Mob. Inf. Syst.*, 1(1):41–80, 2005.

[45] Jennica Falk, Peter Ljungstrand, Staffan Björk, and Rebecca Hansson. Pirates: proximity-triggered interaction in a multi-player game. In *CHI '01: CHI '01 extended abstracts on Human factors in computing systems*, pages 119–120, New York, NY, USA, 2001. ACM.

[46] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[47] Vihang Garg, Raman Adaikkalavan, and Sharma Chakravarthy. chapter Extensions to Stream Processing Architecture for Supporting Event Processing, pages 945–955. 2006.

[48] D. Garlan, D. P. Siewiorek, A. Smailagic, and P. Steenkiste. Project aura: toward distraction-free pervasive computing. *Pervasive Computing, IEEE*, 1(2):22–31, 2002.

[49] Stella Gatziu and Klaus R. Dittrich. Events in an active object-oriented database system, 1993.

[50] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, New York, NY, USA, 2003. ACM.

[51] Altaf Gilani, Satyajeet Sonune, Balakumar Kendai, and Sharma Chakravarthy. chapter The Anatomy of a Stream Processing System, pages 232–239. 2006.

[52] Object Managment Group. *The Common Object Request Broker: Architecture and Specification, V2.1.* Object Management Group, 1995.

[53] The STREAM Group. Stream: The stanford stream data manager. Technical Report 2003-21, Stanford InfoLab, 2003.

[54] Daniel Gyllstrom, Eugene Wu 0002, Hee-Jin Chae, Yanlei Diao, Patrick Stahlberg, and Gordon Anderson. Sase: Complex event processing over streams. *CoRR*, abs/cs/0612128, 2006.

[55] Mads Haahr and Vinny Cahill. Sister project proposal. January 2004.

[56] Mads Haahr, René Meier, Paddy Nixon, and Vinny Cahill. Filtering and scalability in the eco distributed event model. In *Proc. of the 5th Int. Symposium on Software Engineering for Parallel and Distributed Systems (ICSE/PDSE2000*, pages 83–95, 2000.

[57] Timothy H. Harrison, David L. Levine, and Douglas C. Schmidt. The design and performance of a real-time corba event service. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 184–200, New York, NY, USA, 1997. ACM.

[58] R.J. Hayton, J.M. Bacon, and K. Moody. Access control in an open distributed environment. *Security and Privacy, IEEE Symposium on*, 0:0003, 1998.

[59] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *In Architectural Support for Programming Languages and Operating Systems*, pages 93–104.

[60] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. *SIGPLAN Not.*, 35(11):93–104, 2000.

[61] Annika Hinze, Kai Sachs, and Alejandro Buchmann. Event-based applications and enabling technologies. In *DEBS '09: Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, pages 1–15, New York, NY, USA, 2009. ACM.

[62] Aleri Inc. http://www.aleri.com/, Oct. 2009.

[63] Apple Inc. Apple announces over 100,000 apps now available on the app store. http://www.apple.com/pr/library/2009/11/04appstore.html, Nov. 2009.

[64] Apple Inc. Apple's app store downloads top two billion. http://www.apple.com/pr/library/2009/09/28appstore.html, Sep. 2009.

[65] Apple Inc. http://www.apple.com/iphone/, Oct. 2009.

[66] Coral8 Inc. Coral8. http://www.coral8.com/, Oct. 2009.

[67] EsperTech Inc. Esper - compex event processing. http://esper.codehaus.org/, Oct. 2009.

[68] GolfTek Inc. Golftek indoor golf simulator and swing analyzer products. http://www.golftek.com/, Jan. 2010.

[69] Google Inc. Developing on a device | android developers. http://developer.android.com/guide/developing/device.html, Feb. 2010.

[70] Gumstix inc. Gumstix connex - feature overview. Last Accesed 23/09/09.

[71] Innovative Sports Training Inc. Swingtrainer. http://www.innsport.com/Sport, Jan. 2010.

[72] RuleCore Inc. Rulecore. http://www.rulecore.com/, Oct. 2009.

[73] StreamBase Systems Inc. http://www.streambase.com/, Oct. 2009.

[74] ZeroC Inc. Ice for android. http://www.zeroc.com/labs/android/index.html, Dec 2009.

[75] Vision IQ. http://poseidon-tech.com/us/index.html. Last accessed, 13/05/2008.

[76] Gehani Jagadish, N. H. Gehani, H. V. Jagadish, and O. Shmueli. Event specification in an active object-oriented database. In *In Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 81–90, 1992.

[77] Chaiporn Jaikaeo, Chavalit Srisathapornphat, and Chien-Chung Shen. Querying and tasking in sensor networks. *Digitization of the Battlespace V and Battlefield Biomedical Technologies II*, 4037(1):184–194, 2000.

[78] Qingchun Jiang, R. Adaikkalavan, and S. Chakravarthy. Mavestream: Synergistic integration of stream and event processing. pages 29–29, July 2007.

[79] Qingchun Jiang, Raman Adaikkalavan, Qingchun Jiang, Raman Adaikkalavan, and Sharma Chakravarthy. Estreams: Towards an integrated model for event and stream processing, 2004. Technical report, 2004.

[80] D. J. Johnston, M. Fleury, and A. C. Downton. An event-based execution model for efficient image processing on workstation clusters and the grid. *Pattern Recognition, International Conference on*, 1:732–735, 2004.

[81] Jens Jorgensen and Soren Christensen. chapter Executable Design Models for a Pervasive Healthcare Middleware System, pages 25–40. 2002.

[82] Balakumar Kendai and Sharma Chakravarthy. Load shedding in mavstream: Analysis, implementation, and evaluation. In *BNCOD '08: Proceedings of the 25th British national conference on Databases*, pages 100–112, Berlin, Heidelberg, 2008. Springer-Verlag.

[83] Cory D. Kidd, Robert Orr, Gregory D. Abowd, Christopher G. Atkeson, Irfan A. Essa, Blair Macintyre, Elizabeth D. Mynatt, Thad Starner, and Wendy Newstetter. The aware home: A living laboratory for ubiquitous computing research. In *CoBuild '99: Proceedings of the Second*

International Workshop on Cooperative Buildings, Integrating Information, Organization, and Architecture, pages 191–198, London, UK, 1999. Springer-Verlag.

[84] I. Korhonen and J.E. Bardram. Guest editorial introduction to the special section on pervasive healthcare. *Information Technology in Biomedicine, IEEE Transactions on*, 8(3):229–234, Sept. 2004.

[85] Jürgen Krämer and Bernhard Seeger. Pipes: a public infrastructure for processing and exploring streams. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 925–926, New York, NY, USA, 2004. ACM.

[86] Guoli Li and Hans arno Jacobsen. Composite subscriptions in content-based publish/subscribe systems. In *In ACM/IFIP/USENIX 6th International Middleware Conference*, pages 249–269, 2005.

[87] C. Liebig, M. Cilia, and A. Buchmann. Event composition in time-dependent distributed systems. In *In CoopIS*, pages 70–78. IEEE Computer Press, 1999.

[88] M. Liljedahl., S. Lindberg, and J. Berg. Digiwall - an interactive climbing wall. In *International Conference on Advances in Computer Entertainment Technology (ACE 2005)*, 2005.

[89] G. Liu, A.K. Mok, and E.J. Yang. Composite events for network event correlation. In *Integrated Network Management, 1999. Distributed Management for the Networked Millennium. Proceedings of the Sixth IFIP/IEEE International Symposium on*, pages 247–260, 1999.

[90] Hawk-Eye Innovations Ltd. Hawk-eye innovations. http://www.hawkeyeinnovations.co.uk/, Jan. 2010.

[91] Kun lung Wu, Philip S. Yu, and Ling Liu. Adaptive load shedding for windowed stream joins. In *In Proc. Int. Conf. on Information and Knowledge Management (CIKM*, pages 171–178, 2005.

[92] Chaoying Ma and Jean Bacon. Cobea: A corba-based event architecture. In *in Proceedings of the 4 rd Conference on Object-Oriented Technologies and Systems, USENIX*, pages 117–131, 1998.

[93] Samuel Madden and Michael J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data, 2001.

[94] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. The design of an acquisitional query processor for sensor networks. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 491–502, New York, NY, USA, 2003. ACM.

[95] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

[96] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. Wireless sensor networks for habitat monitoring. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 88–97, New York, NY, USA, 2002. ACM.

[97] Masoud Mansouri-Samani and Morris Sloman. Gem: a generalized event monitoring language for distributed systems. *Distributed Systems Engineering*, 4(2):96–108, 1997.

[98] Brian Babcock Mayur, Brian Babcock, Mayur Datar, and Rajeev Motwani. Load shedding techniques for data stream systems. In *In Proc. of the 2003 Workshop on Management and Processing of Data Streams (MPDS*, 2003.

[99] René Meier and Vinny Cahill. Steam: Event-based middleware for wireless ad hoc network. In *ICDCSW '02: Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 639–644, Washington, DC, USA, 2002. IEEE Computer Society.

[100] Rene Meier and Vinny Cahill. Exploiting proximity in event-based middleware for collaborative mobile applications. In *In Proceedings of the 4th IFIP International Conference on Distributed Applications and Interoperable Systems (DAISâĂŹ03). Springer-Verlag*, pages 285–296. Springer-Verlag, 2003.

[101] Rene Meier and Vinny Cahill. Taxonomy of distributed event-based programming systems. *Comput. J.*, 48(5):602–626, 2005.

[102] Sun Microsystems. Rpc: Remote procedure call protocol specification, 1988.

[103] Sun Microsystems. *Javabeans api sepcification, version 1.01*, July 1997.

[104] Sun Microsystems. *Java Distributed Event Specification*, 1998.

[105] Sun Microsystems. Sunspotworld. http://www.sunspotworld.com/, Oct. 2009.

[106] Robert Munro. Smartphone sales are holding up. http://www.theinquirer.net/inquirer/news/1561143/smartp sales-holding, Nov. 2009.

[107] Joanna Alicja Muras, Vinny Cahill, and Emma Katherine Stokes. A taxonomy of pervasive healthcare systems. In *Pervasive Health Conference and Workshops, 2006*, pages 1–10, 29 2006-Dec. 1 2006.

[108] ndrc. Viking ghost hunt. www.ndrc.ie/portfolio/entertainment/viking-ghost-hunt, Sep. 2010.

[109] Nokia. N96. http://europe.nokia.com/find-products/devices/nokia-n96/main/landing, Oct. 2009.

[110] Karl O'Connell, Tom Dinneen, Steven Collins, Brendan Tangney, Neville Harris, and Vinny Cahill. Techniques for handling scale and distribution in virtual worlds. In *EW 7: Proceedings of the 7th workshop on ACM SIGOPS European workshop*, pages 17–24, New York, NY, USA, 1996. ACM.

[111] P3ProSwing. P3proswing. Last accessed 29 June 2005.

[112] Peter R. Pietzuch and Jean M. Bacon. Hermes: A distributed event-based middleware architecture. *Distributed Computing Systems Workshops, International Conference on*, 0:611, 2002.

[113] Peter R. Pietzuch, Brian Shand, and Jean Bacon. A framework for event composition in distributed systems. In *Middleware '03: Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, pages 62–82, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[114] P.R. Pietzuch, B. Shand, and J. Bacon. Composite event detection as a generic middleware extension. *Network, IEEE*, 18(1):44–55, Jan/Feb 2004.

[115] S. Reilly, P. Barron, V. Cahill, K. Moran, and M. Haahr. *Digital Sport for Performance Enhancement and Competitive Evolution: Intelligent Gaming Technologies*, chapter A General-Purpose Taxonomy of Computer-Augmented Sports Systems. IGI Global, Hershey, PA, 2009.

[116] P. Remagnino and G. L. Foresti. Ambient intelligence: A new multidisciplinary paradigm. *Systems, Man and Cybernetics, Part A, IEEE Transactions on*, 35(1):1–6, 2005.

[117] Swing Revolution. Swing revolution. Last accessed 1st July 2005.

[118] D M Ritchie, S C Johnson, M E Lesk, and B W Kernighan. The c programming language. pages 85–113, 1986.

[119] Antony I. T. Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. Scribe: The design of a large-scale event notification infrastructure. In *NGC '01: Proceedings of the Third International COST264 Workshop on Networked Group Communication*, pages 30–43, London, UK, 2001. Springer-Verlag.

[120] Stan Zdonik Sbz, Stan Zdonik, Michael Stonebraker, Mitch Cherniack, Ugur Ceintemel, Magdalena Balazinska, and Hari Balakrishnan. The aurora and medusa projects. *IEEE Data Engineering Bulletin*, 26, 2003.

[121] Robert W. Scheifler and James Gettys. *X Window system: core and extension protocols*. Digital Equipment Corp., Acton, MA, USA, 1997.

[122] Scarlet Schwiderski-Grosche. Context-dependent event detection in sensor networks. Fast abstract on DEBS'08, 2008.

[123] Scarlet Schwiderski-Grosche. Spatio-temporal reasoning with composite events in mobile systems. Fast abstract on DEBS'08, 2008.

[124] Scarlet Schwiderski-Grosche and Ken Moody. The spatec composite event language for spatio-temporal reasoning in mobile systems. In *DEBS '09: Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, pages 1–12, New York, NY, USA, 2009. ACM.

[125] Mohamed A. Sharaf, Panos K. Chrysanthis, Alexandros Labrinidis, and Kirk Pruhs. Efficient scheduling of heterogeneous continuous queries. In *In The International Journal on Very Large Data Bases (VLDB J*, 2006.

[126] Chien-Chung Shen, C. Srisathapornphat, and C. Jaikaeo. Sensor information networking architecture and applications. *Personal Communications, IEEE*, 8(4):52–59, Aug 2001.

[127] Andrew W. B. Smith and Brian C Lovell. Autonomous sports training from visual cues. In *Australian and New Zealand Intelligent Information Systems*, 2003.

[128] Chavalit Srisathapornphat, Chaiporn Jaikaeo, and Chien-Chung Shen. Sensor information networking architecture. *Parallel Processing Workshops, International Conference on*, 0:23, 2000.

[129] Gradimir Starovic, Vinny Cahill, and Brendan Tangney. An event based object model for distributed programming. In *Dublin City University*, pages 72–86. Springer-Verlag, 1995.

[130] Richard W. Stevens. *Unix Network Programming*. Prentice Hall PTR, January 1990.

[131] Sun. Java messaging service. http://java.sun.com/products/jms/.

[132] Information Society Technologies. Embedded systems - facts and figures. http://cordis.europa.eu/ist/embedded/facts_figures.htm, Dec. 2006.

[133] Crossbow Technology. Crossbow technology. http://www.xbow.com/, Sep. 2009.

[134] Monica Tentori and Jesus Favela. Activity-aware computing for healthcare. *IEEE Pervasive Computing*, 7(2):51–57, 2008.

[135] Upkar Varshney. Pervasive healthcare and wireless health monitoring. *Mobile Networks and Applications*, 12(2):113–127, Jun 2007.

[136] Mark Weiser. The computer for the 21st century. *Scientific American*, 265(3):94–104, sep 1991. http://www.ubiq.com/hypertext/weiser/SciAmDraft3.html.

[137] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 407–418, New York, NY, USA, 2006. ACM.

[138] Yong Yao and Johannes Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Rec.*, 31(3):9–18, 2002.

[139] Detlef Zimmer, Axel Meckenstock, and Rainer Unland. A general model for event specification in active database management systems. In *In Proceeding 5th DOOD*, pages 419–420, 1997.

Appendix A

Results of Experiments

Event Frequency (Hz)	Total CPU Time (s)	Total Time(s)	CPU Utilisation(%)
2.96	100	300.36	0.33
2.96	103	300.34	0.34
2.96	100	300.32	0.33
11.89	392	300.34	1.31
11.89	388	300.33	1.29
11.85	394	300.32	1.31
29.37	977	300.35	3.25
29.37	975	300.33	3.25
29.37	981	300.33	3.27
71.41	2383	300.39	7.93
71.37	2375	300.39	7.91
71.54	2376	300.38	7.91
136.29	4522	300.47	15.05
136.25	4529	300.47	15.07
136.42	4512	300.46	15.02
249.48	8207	300.78	27.29
249.44	8207	301.79	27.28
246.2	8205	300.87	26.91

Table A.1: CPU Utilisation C-AESOP

Event Frequency	Total CPU Time (s)	Total Time (s)	CPU Utilisation
2.94	.88	300.37	.29
2.94	.89	300.38	.3
2.94	.90	300.37	.3
11.88	3.57	300.36	1.19
11.88	3.56	300.38	1.18
11.88	3.58	300.37	1.19
29.37	8.82	300.37	2.93
29.4	8.80	300.39	2.93
29.4	8.80	300.39	2.92
71.37	20.98	300.48	6.98
71.4	21.30	300.44	7.09
71.4	21.40	300.44	7.12
136.26	40.84	300.54	13.59
136.26	40.60	300.54	13.51
136.26	40.61	300.54	13.51
249.6	74.67	300.82	24.82
249.57	74.57	300.85	24.781
249.54	74.77	300.91	24.85

Table A.2: Steam CPU Utilisation

Event Frequency (Hz)	Total Time	CPU time	CPU Utilisation	RAM
3.96	565	149	.26	2.087
3.69	314	89	.28	2.087
3.96	313	97	.31	2.087
9.74	347	262	.76	2.087
9.72	288	223	.77	2.087
9.77	267	207	.76	2.087
23.37	537	783	1.45	2.087
23.36	488	717	1.46	2.087
23.36	317	469	1.47	2.087
48.21	381	936	2.46	2.087
48.15	401	942	2.35	2.087
48.04	260	656	2.52	2.087
92.7	266	1194	4.48	2.087
92.91	319	1407	4.41	2.087
93.07	302	1339	4.43	2.087
174.46	260	2143	8.24	2.087
172.55	255	2213	8.68	2.087
174.16	221	1811	8.19	2.087

Table A.3: Custom Event System Performance

Event Frequency (Hz)	Total Time	CPU time	CPU Utilisation	RAM
3.99	303	108	.36	2.09
3.98	265	92	.35	2.09
3.98	330	118	.36	2.09
9.74	270	233	.86	2.09
9.74	310	263	.85	2.09
9.74	280	222	.85	2.09
23.42	302	558	1.85	2.09
23.47	479	883	1.84	2.09
23.45	313	588	1.88	2.09
47.79	457	1350	2.954	2.09
47.62	267	790	2.96	2.09
47.81	281	826	2.94	2.09
92.15	237	1347	5.68	2.09
92.04	256	1451	5.67	2.09
92.01	314	1777	5.66	2.09
170.92	380	3950	10.39	2.09
172.142	126	1312	10.41	2.09
171.96	321	3335	10.39	2.09

Table A.4: J-AESOP Performance

Appendix B

Additional Source Code

Listing B.1: The Fall Detection Example Application in C-AESOP

```
1 #include "steam_interface.h"
2 #include "libmeh.h"
3 #include "vector.c"
4 #include "acceleration.h"
5 #include "emergency.h"
6 #define ACCELEROMETER_HIP "ACCELEROMETER_HIP"
7 #define ACCELEROMETER_WRIST "ACCELEROMETER_WRIST"
8 #define ACCELEROMETER_SHOE "ACCELEROMETER_SHOE"
9 #include <time.h>
10 #include <stdio.h>

12 void parse_event_1(event *);  //this is my callback function for STEAM need one
       for each input stream
13 void parse_event_2(event *);
14 void parse_event_3(event *);

16 int execution_policy();
17 stream* inputs[3];
18 struct tm *local;
19 time_t t;
20 void handler(accel_vector* hip, accel_vector* wrist,accel_vector* shoe,
       emergency* result){

22   struct accel_vector_stream_entry *np;
23   struct accel_vector_listhead *head = ((accel_vector_stream*)inputs[0]) ->
         head;
```

```
24    for (np = head->tqh_first; np != NULL; np = np->entries.tqe_next){
25      if(magnitude(&(np->value),3) > 15.0){
26        t = time(NULL);
27        local = localtime(&t);
28        strcpy(result -> time ,asctime(local));    //assigns the current time to
              the emergency
29        return;
30      }
31    }
32    head = ((accel_vector_stream*)inputs[1]) -> head;
33    for (np = head->tqh_first; np != NULL; np = np->entries.tqe_next){
34      if(magnitude(&(np->value),3) > 15.0){
35        t = time(NULL);
36        local = localtime(&t);
37        strcpy(result -> time ,asctime(local));    //assigns the current time to
              the emergency
38        return;
39      }
40    }
41    head = ((accel_vector_stream*)inputs[2]) -> head;
42    for (np = head->tqh_first; np != NULL; np = np->entries.tqe_next){
43      if(magnitude(&(np->value),3) > 15.0){
44        t = time(NULL);
45        local = localtime(&t);
46        strcpy(result -> time ,asctime(local));    //assigns the current time to
              the emergency
47        return;
48      }
49    }
50  }

52  multi_event_handler* fall_detection_meh;

54  int main(){

56    setup_steam();
57    stream* output =  (stream*) create_swing_output_stream(1,EMERGENCY_SUBJECT);
58    fall_detection_meh =create_multi_event_handler(3,inputs,output,handler,
          execution_policy);
59    inputs[0] = (stream*) create_accel_vector_input_stream(10,ACCELEROMETER_HIP,
          parse_event_1,fall_detection_meh);
60    inputs[1] = (stream*) create_accel_vector_input_stream(10,ACCELEROMETER_WRIST
```

```
        ,parse_event_2,fall_detection_meh);
61    inputs[2] = (stream*) create_accel_vector_input_stream(10,ACCELEROMETER_SHOE,
        parse_event_3,fall_detection_meh);

63    while(1){
64      sleep(1000);
65    }
66  }

68  //checks the hip accelerometer output to see if the patient has changed posture
69  int execution_policy(){
70    printf("Call EP \n");
71    struct accel_vector_stream_entry *latest;
72    struct accel_vector_stream_entry *previous;
73    float threshold = 1; //radian value for threshold for when we decide the
        person is lying down
74    accel_vector upright = {0,-9.8,0};
75    if(!inputs[0] -> status)
76      return 0;      //if the hip accelerometer has not got a new reading

78    struct accel_vector_listhead *head = ((accel_vector_stream*)inputs[0]) ->
        head;
79    latest = head->tqh_first ;
80    previous = latest ->entries.tqe_next;
81    if(previous == NULL){
82      //only one value, no change in posture
83      return 0;
84    }
85    if(angle_between(&(latest -> value) , &upright  , 3) > threshold  &&
86       angle_between(&(previous -> value) , &upright  , 3) < threshold ){
87      printf("Fire Multi-Event Handler \n");
88      return 1;  //change in posture detected
89    }
90  }



93  void parse_event_1(event *ev){

95    parse_accel_vector_event(ev,inputs[0]);

97  }
```

```
99 void parse_event_2(event *ev){
100   parse_accel_vector_event(ev,inputs[1]);
101 }
102 void parse_event_3(event *ev){
103   parse_accel_vector_event(ev,inputs[2]);
104 }
```

Listing B.2: The Fall Detection Example Application in J-AESOP

```
1  package ie.ndrc.vgh.AESOP;

3  import java.util.Iterator;
4  import java.util.List;

6  public class FallDetection extends MultiEventHandler {

8          EventStream s1;
9          EventStream s2;
10         EventStream s3;
11         EventManager manager = EventManager.getInstance();
12         double threshold = 60;

14         public FallDetection() {
15                 Event e1 = new Event("ACCEL_HIP");
16                 Event e2 = new Event("ACCEL_ANKLE");
17                 Event e3 = new Event("ACCEL_WRIST");
18                 s1 = new EventStream(10, e1.type, this);
19                 s2 = new EventStream(10, e2.type, this);
20                 s3 = new EventStream(10, e2.type, this);
21         }

23         public boolean executionPolicy() {
24                 Iterator<Event> acceleration_1 = s1.getWindow().iterator();
25                 MathVector upright_accel_vector = new MathVector(0, -9.8, 0);
26                 if (MathVector.angleBetween(acceleration_1.next(),
                       upright_accel_vector) > threshold) {
27                         if (acceleration_1.hasNext()
28                                         && MathVector.angleBetween(
                                                acceleration_1.next(),
29                                                        upright_accel_vector) <
                                                             threshold) {
```

```
30                              return true; // posture has changed from
                                    vertical to horizontal
31                      }
32              }
33              return false; // no change in posture

35      }

37      public void handler() {
38              Iterator<Event> acceleration_1 = s1.getWindow().iterator();
39              Iterator<Event> acceleration_2 = s2.getWindow().iterator();
40              Iterator<Event> acceleration_3 = s3.getWindow().iterator();

42              while (acceleration_1.hasNext()) {
43                      if (vectorMagnitude(acceleration_1.next()) > 15) {
44                              call_ambulance();
45                              return;
46                      }
47              }
48              while (acceleration_2.hasNext()) {
49                      if (vectorMagnitude(acceleration_2.next()) > 15) {
50                              call_ambulance();
51                              return;
52                      }
53              }
54              while (acceleration_3.hasNext()) {
55                      if (vectorMagnitude(acceleration_3.next()) > 15) {
56                              call_ambulance();
57                              return;
58                      }
59              }

61      }

63      private void call_ambulance() {
64              // TODO Auto-generated method stub

66      }

68      private int vectorMagnitude(Event next) {
69              // TODO Auto-generated method stub
70              return 0;
```

```
71              }

73  }
```

Appendix C

CD of Source Code