PiCSE: A Framework for Simulation and Emulation of Pervasive Computing Applications

Vincent Reynolds

A thesis submitted to the University of Dublin, Trinity College

in fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

March 2015

Declaration

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

Vincent Reynolds

Dated: 9th March 2015

Permission to Lend and/or Copy

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

Vincent Reynolds

Dated: 9th March 2015

Acknowledgements

" Oh codswallop! It's taken me eleven years, and it's perfect. "Edmund: A Butler's Tale" – a giant roller-coaster of a novel in four hundred sizzling chapters. A searing indictment of domestic servitude in the eighteenth century, with some hot gypsies thrown in. My magnum opus, Baldrick. Everybody has one novel in them and this one's mine",

Edmund Blackadder, c 1793.

I have had the pleasure of working with many colleagues who have become some of my closest friends. In particular, Anthony, Daire, Ray, Andronikos and Melanie have supported me when I needed it and that means a lot to me, more than I can express in these short words. There's far too many others to list all the names, but thanks for all the coffees, the football, the cakes, the biscuits, the Starcraft and the beers.

I owe a special debt of gratitude to my supervisor, Vinny Cahill. His guidance and patience throughout the years have kept the ship steady, and I am indebted to him for all of the opportunities which have arisen out my time in the DSG.

Finally, out in the real world, I wouldn't be where I am today without the support and love of Ba and my daughter Maya. Obrigada amores.

Vincent Reynolds

University of Dublin, Trinity College

March 2015

Abstract

The field of pervasive computing is changing rapidly, with the proliferation of and advances in hardware and communication technologies resulting in a shift towards a large-scale, heterogeneous pervasive computing world where people and devices move seamlessly between previously isolated pervasive computing applications. Prototyping and evaluation of pervasive computing scenarios is a necessary yet inherently difficult step between design and deployment of new applications, and simulation has traditionally been both an effective and accepted modelling and evaluation methodology that addresses this step.

Previous efforts to provide generic simulation tools that fulfill the unique requirements of pervasive computing have met only limited success and these have typically focused on extending or modifying existing tools that have been designed for other purposes. There are several existing simulators dedicated to specific pervasive computing domains such as wireless sensor networking, context-aware computing, smart spaces, or intelligent transportation systems. Additionally there are many emulators that provide platform-specific application programming interfaces for the evaluation of pervasive applications in these domains. Pervasive computing now encompasses so many sub-domains, technologies, and disciplines that perhaps it is no surprise that truly generic pervasive computing simulators have yet to be created. Previous work generally suffers from one of two problems: simulators target a specific application platform or a sub-domain of pervasive computing, with the result that substantial modifications are required to either simulate or integrate simulations of other sub-domains; or, generic simulators that are too abstract require substantial effort in re-creating recurring elements when specializing the simulator to a particular domain.

The resulting systems do not meet the modelling and evaluation requirements of large-

scale, heterogeneous pervasive computing applications so there remains a significant open research challenge in this area, which is to investigate whether a generic simulator can be developed that can model a broad range of existing pervasive applications as well as new applications that may emerge in the future. The hypothesis of this thesis is that a framework-based approach to building generic integrated pervasive computing simulators and emulators is a sufficiently flexible and extensible approach for the evaluation and modelling of pervasive applications.

The use of frameworks is a well-recognised software engineering paradigm that is appropriate when flexibility, extensibility and reusability are required from the outset. A framework comprises a set of abstract classes that capture recurring patterns within a group of related problems.

The contribution of this thesis is to show that using a framework is a sufficiently flexible and extensible approach to creating simulators and simulations for a broad range of pervasive computing domains. The PiCSE framework - the Pervasive Computing Simulation and Emulation Environment - identifies the most common abstractions found in this field, such as sensors and the physical environment, and provides an architecture for building simulators for the many different pervasive computing sub-domains that exist. These simulators can then be used to create specific simulations of particular scenarios for these sub-domains. Furthermore, the framework can be used to provide an emulation environment for applications, and mediate the interaction between those emulated applications and any simulated components.

This thesis describes the design and implementation of the PiCSE framework and shows how this approach addresses the research challenge in this area. The contribution is evaluated by instantiating three scenarios meeting different requirements such as scale, use of diverse hardware artefacts, and complex application execution environments. These instantiations validate the framework's modularity, flexibility and suitability as an approach for the creation of simulators and simulations for the modern pervasive computing domain.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Simulation can play an important role in the life-cycle of software engineering projects (Zeigler 00). This is particularly true in the case of pervasive computing where the scale of applications can be large and they are often designed to be deployed in less than ideal physical environments (Razafindralambo 10). Designing a simulator that meets the modelling requirements of pervasive computing is a difficult challenge arising from three factors: there is a broad diversity of application areas within the domain; these application areas are now intersecting, and finally, new application areas are emerging quickly. At present, new simulation tools are being developed to keep pace with each new application area (Mangharam 06; Eugster 06; Jouve 09) within the evolving pervasive computing domain. This thesis describes the Pervasive Computing Simulation and Emulation (PiCSE)[1] framework, its architecture and design, the combination of which addresses these challenges.

PiCSE is a framework (Johnson 88) that supports both the creation of pervasive computing simulators and individual simulations through the modelling of recurring pervasive computing abstractions and their relationships. The creation of domain-specific pervasive computing simulators such as intelligent transportation system (ITS) simulators or smart-space simulators is enabled by specialising the PiCSE framework's abstractions to create domain-specific abstractions.

The instantiation of a group of PiCSE abstractions, or of any specialised domain-specific

[1] Pronounced as Pixie

abstractions provided by a domain-specific simulator results in the creation of an *instance simulation*: a simulated model of a specific pervasive computing scenario. Moreover, the PiCSE framework supports the emulation of applications that form an integral part of these scenarios, and these emulated applications can be integrated into instance simulations. The PiCSE simulation engine, which underpins the framework, mediates the interaction of emulated applications with simulated artefacts. PiCSE's flexible approach enables a person to instantiate the framework to create domain-specific simulators and simulations to evaluate a diverse range of pervasive computing application areas.

The remainder of this chapter is structured as follows. Section 1.1 provides a broad introduction to the area of pervasive computing. The motivation and current methodologies for evaluating pervasive computing applications are presented in section 1.2 which then draws on a discussion of these approaches in the field to derive the key challenges in this area. Section 1.3 introduces the concept of frameworks and suggests why a framework-driven approach might be successful in addressing these challenges. A description of the contribution of the thesis and the evaluation methodology used is provided in section 1.4 and a road map of the remainder of the thesis concludes this chapter.

1.1 Pervasive Computing

Pervasive computing (Weiser 91; Weiser 93) as an area of research is one that encompasses more focused areas such as ambient intelligence (Bresciani 04), sensor networking (He 04), context-aware computing (Benerecetti 01), and smart spaces (Dearle 03). Consider the following pervasive computing applications. Smart-spaces (Tapia 04) are often passive applications, acting in the background and driven by local sensors, which can then adapt some aspect of the space such as access control (Selvarajah 10) or telephone call re-routing to a user's requirements. An environmental-monitoring wireless sensor network (Ji 04) could comprise sensors, perhaps monitoring precipitation, noise, or temperature, attached to wireless sensor motes that propagate that sensor information to other sensor motes through an ad-hoc communication network. Intelligent transportation systems (Klein 01) typically use embedded sensors such as inductive loops in the road network as inputs into algorithms (Salim 08) that improve

the throughput of the vehicles in the system. Despite the diversity of these applications, all of them can be considered to lie within the broad domain of pervasive computing.

More recently, there has been a shift towards the integration of heterogeneous pervasive computing applications. As adoption and deployment of pervasive computing technologies takes hold, the physical and logical boundaries between these applications are being reduced and now overlap in many cases (Handte 09). The traditional notion of a pervasive computing application as a closed, isolated system is being eroded with the integration of service-oriented architectures into pervasive computing applications (Guinard 10), enabling users to exploit the services and functionality of different pervasive computing applications as they move between them. If the integration of these pervasive applications is to be successful, information describing users of these applications and the applications themselves must be freely available and understandable by all so that it can be consumed by all pervasive computing applications when and where they need it. As a result, there has been attempts recently towards centralising these services: two recent examples are Google Latitude[2] which provides services for managing a user's location and Conserv (Hynes 09) which manages a wider spectrum of a user's contextual data such as their calendar events and personal devices that might identify them.

In addition, a new class of ad-hoc pervasive computing is now emerging with the user at its centre. In citizen sensing (Campbell 08) or participatory sensing as it is sometimes known, users with powerful sensor-enabled mobile devices are augmenting, and in some cases replacing, static embedded sensors in the physical environment. The "wisdom of crowds", or their collective sensed information in this case, is leading to innovative crowd-sourced approaches to environmental monitoring (Lu 09) and even supporting the creation of interactive art installations (Vyas 10).

These examples represent a small sub set of all the domains that pervasive computing address, yet despite the diverse nature of these pervasive computing sub-domains, there are a set of recurring characteristics that are common within them. They are typically deployed in sensor-equipped spaces, whether those sensors are static or mobile, where information regard-

[2] www.google.com/latitude

ing the environment and local contextual information is leveraged by applications to provide more tailored and contextualised functionality and services within that environment. In addressing this area, pervasive computing applications have to contend with many non-trivial problems including dependability, large scale, physical distribution, security and timeliness. Moreover, typical environments in which these applications are deployed include buildings for smart-space scenarios (Tapia 04), rugged terrain for environmental monitoring (Ji 04) and road networks for intelligent transportation systems (Klein 01). These environments are often physically in a remote location making deployment and maintenance both difficult and costly.

1.2 Motivation

Modelling and evaluation plays a critical role in the life-cycle of any software engineering project but presents some particular challenges in pervasive computing application scenarios. The application typically extends beyond the computer into the physical environment through the use of sensors, actuators, or location-based services. The implications for the testing of these scenarios is that the complete pervasive computing eco-system, including the environment, devices, their users and other factors should be considered. However, this introduces difficulties that must be overcome. Testing in physical environments such as ocean floors (Jiang 09), offices and homes (Selvarajah 10) to road networks (Salim 08) and sports environments (Kranz 07) can be invasive, disruptive and often hampered by bureaucracy, if access is even available. Many of these applications are predicated upon hardware technologies such as cheap, reliable sensors, for example, radio frequency identification tags (Xu 10) or wireless sensors (Medagliani 10), which are yet to be widely deployed or even available in the scales that are envisioned, and as a result, accurate real-world deployments in some domains may not yet be feasible. Pervasive computing applications in the areas of intelligent transportation systems or urban-sensing for example, can involve the interaction of thousands or tens of thousands of elements. Finally, the evolving nature of the field means that new areas are emerging and the pervasive computing community currently lacks the appropriate tools to evaluate some of these areas such as participatory-sensing (Campbell 08). In these cases, proof-of-concept prototypes or custom evaluation tools may have to be developed. Factors

such as these impact upon the time and effort required to evaluate and debug a pervasive computing prototype, particularly when considering that an exhaustive set of controlled scenarios may have to be deployed in order to evaluate a prototype rigorously.

1.2.1 Current Approaches to Evaluating Pervasive Computing Applications

(Nakata 07) lists testbeds, small-scale laboratory tests and simulation as three methodologies for evaluating pervasive computing applications. Which of these methodologies is chosen to evaluate any particular pervasive computing scenario is influenced by experimental factors such as the application's current stage of development and on what the desired outcome of the evaluation is. The methodology chosen can additionally be influenced by local factors such as the tester's access to hardware devices and testing facilities, costs, and the time that is available to test the application. If the purpose of the evaluation is to validate whether a pervasive computing application is robust enough to be ready for a commercial release and subsequent real-world deployment, then an appropriate evaluation methodology might be to use a large-scale testbed, whereas in the creation of an application prototype in an emerging large-scale field, a simulation-based evaluation may be used.

Physical testbeds such as the Twist (Handziski 06) and MoteLab (Werner-Allen 05) testbeds provide an experimental platform in which applications are executed on hardware devices in a stable environment, where there is usually a supporting infrastructure in the form of back-channels and monitors for the management, controlled execution and debugging of experiments. Using native hardware devices as the platform for an experiment potentially gives greater credibility to results gathered from any experiment when compared, for example, with simulated results however, the methodology and supporting infrastructure usually restricts the testbed to that particular platform and thus to a specific domain of pervasive computing. To date, most testbeds are very domain specific, have not been designed with integration of other domains in mind and thus are not suitable as a generic pervasive computing evaluation platform.

Small scale laboratory testing is a methodology similar to the use of testbeds in that

experiments are executed on real hardware devices. In contrast to testbeds however, there is rarely a permanent infrastructure supporting the experiments, resulting in experiments that are often not very scalable and that can be difficult to reproduce in other laboratories.

Finally, there are several existing pervasive computing simulators but few that recognise or reflect that pervasive computing is becoming a more multi-disciplinary domain with new emerging areas of research and their crossover with existing areas of research. As an evaluation methodology, simulation's popularity is demonstrated by its wide acceptance within the pervasive computing community as a valid methodology, and many simulation libraries are open-source and are available for little or no cost (Fall 01; Martin 06a) so it has a low barrier of entry in this respect. However, in general, these simulators are rarely designed for inter-operation (Kuhl 99) and are not capable of modelling the diverse range of pervasive computing scenarios that exist, i.e., a network simulator is not readily adapted to the modelling of a context-aware computing scenario because it was not intended to model concepts such as user behaviour and activities. Therefore, any simulated scenario that includes both aspects of context-aware computing and wireless networking requires either the creation of a new simulation tool for that particular domain, or the modification and integration of existing tools. In addition to simulation, emulation provides a low cost alternative to the evaluation of applications by removing the physical requirement for the intended device platform to be available for the evaluation of a particular application. Scalable and cost-effective prototyping can be achieved as multiple instances of an emulator can be run in parallel, without the experimental overhead of managing and deploying an application on the hardware devices. Many existing emulators within the pervasive computing domain are written for applications targeted for a single specific platform. For example, the TOSSIM emulator (Levis 03) supports TinyOS (Hill 00) applications only, although these can run on several related hardware platforms such as the TMote Sky[3] and the MoteIV. By constraining an emulator to a single application platform, an emulator is implicitly constraining its scope to a specific domain as these application platforms are generally domain-specific, in this case, wireless sensor networking.

[3] www.sentilla.com

1.2.2 A Set of Challenges for the Modelling and Evaluation of Pervasive Computing Scenarios

This thesis is concerned with methodologies that are both suitable for the evaluation *of a wide range of diverse pervasive computing scenarios* and that *can be extended to evaluate new scenarios* that may emerge in the future.

It can be seen from the approaches that have been described in the preceding sections that testbeds, small-scale laboratory testing, simulators and emulators meet these requirements to varying degrees of success. Testbeds are very effective tools and can provide excellent experimental support, however they are expensive to create and maintain, and are usually constrained to a single application domain and hardware platform. In many domains such as intelligent transportation systems (ITS) and urban sensing, large-scale testbeds may not even be feasible to anyone except the largest industrial research labs. Whilst it is possible to create bespoke small-scale laboratory tests for almost any domain, these are inherently limited in scale, are often built without consideration for extensibility and require physical reconfiguration for each new scenario. In the case of both existing simulators and emulators, there are a wide range of domain-specific tools and libraries in existence which are often extensible within their domain but cannot be easily integrated with each other despite their software-only approach. They additionally address complimentary aspects of pervasive computing scenarios. Simulators can model and be used to evaluate physical elements such as sensor devices, users, a wireless network, and the environment, whilst an emulator can model the run-time environment of applications that execute within those scenarios and potentially interact with those elements. In conclusion, there is no *one size fits all* evaluation strategy that can be applied to pervasive computing applications however, a flexible and combined approach of both simulation and emulation could potentially support the evaluation of large-scale heterogeneous applications.

Based on the issues identified in the overview of pervasive computing domains in section 1.1 and in the range of modelling and testing methodologies used to evaluate those domains in section 1.2.1, the following research problem has been identified. As pervasive computing applications grow in scale and ubiquity, the range of elements that have be taken into

consideration when evaluating those applications is also increasing. However, all of the evaluation methodologies suffer to a certain degree from the same weakness, in that they are tied to a specific pervasive computing domain and are generally not designed to be extended to evaluate new applications which could draw from more than one independent domain.

One of the main arguments for maintaining the status quo of testing individual aspects of particular pervasive computing domains in isolation is the need for rigorousness in the experimental method. By isolating as many experimental factors as possible, it is then easier to achieve quantifiably independent, reproducible and objective results. Certainly, using a single independent simulator is at least theoretically more suitable for this task. However, as has been noted, new multi-disciplinary areas of pervasive computing such as vehicular ad-hoc networks (VANETs) and smart spaces makes it more difficult to treat these areas in isolation. In the case of VANETs(Klein 01), the behaviour model responsible for the movement of a vehicle is a factor in the effectiveness of a routing protocol that might be running within that VANET. Of course, the behavioural model of the vehicle can be reduced to a plug-in or a mobility pattern that might be integrated into a network simulator, but that model still has to be built by an expert in the vehicular community. Furthermore, there may be applications in the future whereby the outcome of an application that is using the routing protocol influences the behaviour of the vehicle. In that case, the limitations of a simple plug-in based approach will be exposed and the integration of both network and vehicle simulators will then have to achieved.

The following challenges have been identified as key issues in addressing this open research problem:

1. **Flexible Heterogeneity:** Any simulator or emulator should be able to support heterogeneous elements, both hardware and software based, and the fact that these elements can interact in a manner that is unrestricted by any adopted approach. These elements may come from different pervasive computing domains.

2. **Extensibility:** Any simulator or emulator should be able to accommodate new emerging areas of pervasive computing not yet identified, where these new areas can comprise of both new hardware and software elements not yet identified.

3. **Reusability:** Any simulator or emulator should simplify the task of building and integrating elements that are to be evaluated.

Simulation and emulation are two evaluation methodologies that can support large-scale scenarios and that are commonly implemented using extensible architectures, albeit generally within their domain only. The hypothesis of this thesis is that a new framework-based approach to building generic integrated simulators and emulators for pervasive computing applications is a sufficiently flexible, extensible, and reusable approach for the evaluation and modelling of these applications. These three challenges are not exhaustive by themselves however. There are additional challenges in this area that include:

1. **Modelling Large-Scale Scenarios:** Many pervasive computing applications require the interaction of many thousands of interacting features such as applications, sensors, and users, and in some cases even more. This motivates the need for a scalable approach in addressing this challenge.

2. **Timeliness:** The realistic modelling of application behaviour in many pervasive computing domains necessitates the accurate modelling of the time taken for an event such as a sensor reading or an application behaviour.

Whilst these are important challenges in the simulation of pervasive computing applications, they are not the key challenges when addressing the issue of the modelling and simulation of overlapping of new and emerging pervasive computing application domains. Nevertheless, these two challenges present secondary challenges and their consideration is addressed in the thesis where appropriate. Finally, application-specific requirements such as *security* and *dependability* provide additional challenges but these are pertinent to the applications that may run within the scenarios and are not specific features that are addressed independently in this study.

1.3 A Framework Approach to Simulating and Emulating Pervasive Computing Applications.

Johnson et al (Johnson 88) define a framework as

> "a set of classes that embodies an abstract design for solutions to a family of related problems."

This definition encapsulates the abstract nature of frameworks, while capturing their suitability to addressing families of related problems. Other definitions, notably those of Campbell (Campbell 91) and Booch (Booch 93) incorporate the concept that the collection of classes constituting a framework, interact in a defined manner or pattern. The benefits of using frameworks are well known, and Schmidt (Schmidt 97) lists these primarily as *modularity*, *reusability* and *extensibility*. He states, that frameworks

> "enhance modularity by encapsulating volatile implementation details behind stable interfaces. This localisation reduces the effort required to understand and maintain existing software."

This definition highlights the use of stable interfaces defined by a framework to abstract from the implementation of the framework. He goes on to state that stable interfaces

> "provided by frameworks enhance reusability by defining generic components that can be reapplied to create new applications. Framework reusability leverages the domain knowledge and prior effort of experienced developers in order to avoid re-creating and re-validating common solutions to recurring application requirements and software design challenges."

This definition of reusability acknowledges that if recurring requirements and software design challenges can be identified amongst a set of related problems, then these can be exploited within a framework to reduce the effort in creating new applications. Finally, Schmidt et al (Schmidt 97) state that a well-designed framework enhances extensibility

> "by providing explicit hook methods that allow applications to extend its stable interfaces."

and that

> "framework extensibility is essential to ensure timely customisation of new application services and features."

The benefits of extensibility are clear within frameworks, as it means that the framework can be extended to support new application features identified in addition to the original set of related application features.

1.3.1 The PiCSE Framework

The PiCSE framework is implemented as a set of abstract classes that can be grouped into two class categories (Kruchten 95), the *PC_ Abstractions* and the *Abstract_ Interfaces* class categories, that address the three key challenges identified; flexible heterogeneity, extensibility, and reusability. A third class category, the *Core_ Components* class category, provides a set of core components, for example, the simulation engine, that underpin and are common, to all simulations created using the PiCSE framework.

PiCSE's *PC_ Abstractions* class category provides the class definitions of recurring elements of pervasive computing applications such as sensors, environments, and application platforms. These abstractions are designed to be sufficiently generic that they may be instantiated to create a diverse range of concrete instances with variable characteristics, thus meeting the requirement of supporting heterogeneity. In addition to this, the abstractions take into consideration the likely interactions between these instantiations, and the framework approach allows concrete instances to be modularly combined, allowing the flexible creation of more complex instantiations.

The framework's *Abstract_ Interfaces* class category comprises a set of abstract interfaces that also underpin the *PC_ Abstractions* class category, allowing for the extension of the framework to create new abstractions. These set of abstractions capture concepts such as mobility, whether an object can be carried, whether a feature can be sensed, and others that

are recurring across pervasive computing domains. This class category addresses the challenge of extensibility within the framework's architecture that can account for emerging domains in the future.

The framework-driven approach addresses the challenge of reusability by providing common solutions to recurring application requirements and software design challenges. All three class categories, the *PC_ Abstractions*, *Abstract_ Interfaces* and *Core_ Components* class categories contribute here by providing the base of an extensible framework, that also captures how these components can potentially interact. The capture of these recurring components and their interactions in a generic way within the framework reduces the effort required to create new simulators and simulations.

1.3.2 Using the PiCSE Framework

The PiCSE framework can be specialised by domain experts to a range of pervasive computing sub-domains. Two experts in separate domains can specialise the PiCSE framework using PiCSE's recurring interaction patterns to create simulators for their respective sub-domains, and that these simulators can themselves be provided in the form of frameworks that can be further specialised where appropriate. Furthermore, simulated and emulated elements of the respective sub-domains can exist and interact within the same simulated space if desired.

1.4 The Thesis

Domain-specific simulators address particular application domains such as wireless sensor networking or context-aware computing and these play an important role in the evaluation of individual application areas. However, their constrained and inflexible architectures do not meet the modelling and evaluation requirements of large-scale, heterogeneous pervasive computing applications so there remains a significant open research challenge in this area, which is to investigate whether a generic simulator can be developed that can model a broad range of existing pervasive applications as well as new applications that may emerge in the future.

This thesis addresses this open research challenge. PiCSE's framework-driven approach and class category implementations will be shown to be sufficiently flexible and extensible that users can create simulators for a broad range of pervasive computing applications that have diverse characteristics, and is a novel contribution to the state of the art in this field.

1.4.1 Implementation

This framework approach is implemented as a set of C++ (Stroustrup 98) libraries, that are grouped into three separate class categories. The simulation engine underlying the system is built upon an existing event-based simulator, Simpack (Fishwick 95), which has been extended using a multi-thread approach to seamlessly and flexibly manage multiple heterogeneous emulated applications and simulated hardware components. Furthermore, the engine supports the interaction of both these simulated hardware components and emulated applications, enabling realistic input and output channels to be created for the emulated applications. Applications that interact directly with an underlying operating system via system calls or alternatively that are built upon middleware can be modelled.

1.4.2 Evaluation

Three instantiations of the framework were created to validate the hypothesis of the thesis's framework-driven approach. In the first *Social-Sensing Scenario*, an event-based middleware is emulated. Applications that are built upon that middleware then exchange messages whilst moving around a constrained physical space. In the second *Car-Hardware Emulation Scenario*, Linux system calls are emulated, and application instances use those system calls to interact with simulated sensors reading data from the environment. The third and final *Intelligent Transportation Systems Scenario* models vehicular traffic in part of Dublin, Ireland and demonstrates the simulator's support for large-scale simulations, a complex environment, and complex inter-related object types.

1.4.3 Thesis Roadmap

The remaining chapters of this thesis provide a detailed description of the PiCSE framework, its contribution to the state of the art and an evaluation of the work. Chapter 2 provides an examination of the state of the art in simulators and emulators for pervasive computing and related domains. Chapter 3 describes the framework, its architecture and design and an additional section within that chapter also describes the framework's support for the emulation of applications. Chapter 4 examines the implementation and features of the key components supported in more detail. An evaluation of the thesis and its contributions is documented in Chapter 5 and the conclusions are presented in Chapter 6.

1.5 Summary

This chapter outlines the motivation for, approach, and contribution of the thesis - the design of a framework that can be specialised to create simulations of pervasive computing scenarios. An introduction to the area of pervasive computing is presented to define the scope of the thesis. The work is then motivated by examining the current methodologies for evaluating pervasive computing scenarios and identifying a set of open challenges. An overview of PiCSE's framework-based approach is then provided before the contribution of the thesis is presented.

Chapter 2

Related Work

2.1 Introduction

The use of simulation as a valid methodology for the modelling and evaluation of pervasive computing applications has been established, and three core challenges, flexibility, extensibility, and reusability have been identified that must be addressed by generic simulators for the broad pervasive computing domain. This thesis describes a modular generic and extensible approach that enables the simulation of a broad range of existing pervasive computing applications as well as new applications that may emerge in the future. The objective of this chapter is to examine established and prior works in this area that have attempted to address these research challenges. In doing so, the strengths and weakness of these approaches are identified and these will be used to derive requirements for the proposed framework-driven approach that forms the contribution to this thesis.

In this chapter, an overview is provided of the many simulators in this field; examining both generic pervasive computing simulators that attempt to address the entire domain, as well as some of the simulators for the many sub-domains of pervasive computing. What is established is that there are few truly generic simulators that take a flexible and extensible approach to modelling the pervasive computing domain as a whole.

This chapter begins by providing a brief overview of some of the classical simulation formalisms and methodologies. Based on these and the challenges identified in chapter one,

a set of evaluation criteria are identified by which the simulators established in this state of the art will be examined. A broad overview of the representative work in this field is then provided, before seven of the most well known and cited simulators representing the state of the art in this space are then examined. The salient points and other notable features of each simulator are noted, and these simulators are measured against the evaluation criteria. Finally, Section 2.6 takes a step back and offers a wider view on the issues raised by the examination of the simulators. Additional simulators are considered at this stage to broaden the overall perspective, and assess alternative approaches and features that are not employed by those examined in detail.

2.2 Model-Based Formalisms and Methodologies

2.2.1 Simulation Formalisms

Simulation is a software-oriented modelling and evaluation methodology and is one of the most popular and widely used within the pervasive computing community today. A simulation is a description of a system, or at least of some of its components expressed in terms of a set of states and events (Zomaya 96). Several simulation formalisms exist (Law 00), most notably the discrete event system specification but also the discrete time system specification, and the differential equation system specification formalisms.

The Discrete Event System Specification

The most popular formalism, the Discrete Event System Specification, known as DEVS (Zomaya 96), specifies a system in terms of a time base, a set of inputs, outputs, current state, and a set of state transition functions. An implementation of a DEVS simulator typically contains a time-ordered list of events, often in the form of function pointers, a clock, some state information, a set of functions for handling events and a loop that advances the simulation. As the clock progresses to the time of the next scheduled event, that event is removed from the event list and any corresponding functionality to that event is invoked. This functionality can have two effects: Firstly, a change in the state information may occur,

and secondly, the event list may be updated, whereby either more events are scheduled in the future or existing future events are cancelled. The challenge in building a successful model or simulation of any system is in the implementation of functionality associated with any processed events.

DEVS, as a formalism, can also be extended to parallel and distributed architectures (Fujimoto 00; Tropper 02). Simulations of complex systems can take a long time on single sequential machines, and opportunities to parallelise these simulations can greatly speed up the execution of these simulations. The amount of actual gain that can be achieved in such systems is determined by a number of factors including the specification of the level of fidelity required and the inter-dependence of any parallelised data sets and processes.

One weakness of the DEVS formalism is that the approach is not easily adapted to real-time simulations, i.e., simulations that appear to proceed at the same speed as the modelled behaviour would appear in the real world. In the DEVS formalism, the simulation steps from event to event, and there is no behaviour modelled or executed in between those events. As, such, there is no modelling of the passing of time in between those events, and the simulation usually completes as quickly as possible. Simulators built using the Discrete Time System Specification formalism take some steps towards addressing this deficiency.

The DEVS model is arguably the most popular model amongst the simulation community at the present time and several conferences[1] and workshops[2], both academic and industrial, are dedicated to solely advancing the model. There are many popular DEVS simulators and tools that are pervasive computing oriented such as ns-2 (Fall 01) and Diasim(Jouve 09). In addition to that, there are several generic simulation tools and libraries, such as C-Sim[3], JavaSim[4], and SimPy which are also available under open source licenses.

The Discrete Time System Specification

In the Discrete Time System Specification (Zomaya 96), a simulation advances at fixed time-steps, and at each time-step the corresponding modelled functionality or behaviour is invoked.

[1] http://www.scs.org/confernc/springsim/springsim09/cfp/springsim09.htm
[2] http://www.isima.fr/~traore/SCSCDEVS/index.html
[3] http://www.c-sim.zcu.cz
[4] http://javasim.codehaus.org

An implementation of this formalism typically contains a clock, some state information, a set of functions that model that state information, and a loop that invokes those functions at each fixed time-step thus advancing the simulation. In the Discrete Time System Specification (DTSS) formalism, all events occur "at the same logical time" and are executed in the order in which they were scheduled, whereas in the DEVS formalism the events are totally ordered globally.

Real-time simulations are a particular variation of the DTSS simulation formalism where the time steps are synchronised with the time of an entity observing the simulation. In this case, the simulation appears to run at what is called the "wall-time". This particular variation has many benefits, most notably that it appears real from a timing perspective to an external observer and is thus suitable for approaches where the observer plays a role in the simulation. This is often the case where there is a graphical interface for the observer in, for example, a training simulator, or a game. This synchronisation with the real world introduces a timeliness requirement of course, which is that all events that must be observed externally must be modelled within that time-step. This can introduce an artificial bound on the number of events that can be simulated given that a simulation is being executed on a device with a fixed number of computing resources.

2.2.2 Emulation

Analysis of application behaviour is an important part of the evaluation of a pervasive computing scenario and is addressed by numerous emulators (Girod 04a; Sobeih 05). An emulator can be defined as a system that implements the *behaviour* of an original system, i.e., a third system cannot differentiate between the behaviour of the original and the emulated system. However due to system resources and factors such as the run-time interpretation of instruction sets, there may be differences in the speed at which the emulated application runs, usually slower. Unlike a simulation which attempts to model the state of a system, an emulator's emphasis is on realistically modelling the interaction between the system and an external party, an application in this case. Emulators allow applications to be executed as if they were actually deployed within their native run-time environment, a requirement that simulators cannot

meet within the overall modelling and evaluation of pervasive computing scenarios. This is important as it allows developers to bypass some of the problems incurred whilst bridging the gap between a simulated algorithm and a real application. The act of implementing and evaluating code twice (Nishikawa 06), once for the simulation and once for the actual implementation, is time consuming and the translation from algorithm to application can introduce differences in application behaviour. Emulation addresses this problem by providing a simulation platform for the evaluation of application code but requires that the target platform is realistically modelled.

There are various approaches to emulating computer programs but these can be split into either hardware- or software-level emulation and the approach that is chosen is usually dependent on the interfaces upon which the emulated program is built. For example, when the emulated application is programmed so that it interacts directly with the hardware, perhaps addressing hardware such as memory, then a hardware emulation technique such as *"binary translation*" can be used. This involves the interpretation and translation of machine instructions from their original form into instructions that the host platform can interpret and execute. The hardware-level approach is also used when the aspect of the program under evaluation is dependent on the hardware being used. For example, within wireless sensor networks, evaluating the performance of a sensor mote can be dependent on the behaviour of the CPU or the radio.

For programs that do not address hardware but perhaps are implemented at a higher logical layer, for example, applications that are built upon system calls, then software emulation is more typically used. In this case, a compatibility layer must be described that provides a mapping from the original system calls to system calls on the host platform. For example, translation software implementations such as Wine (Win 07) and WABI (Sun Microsystems 96) both employ this technique, e.g., allowing Windows programs to run unmodified on x86 UNIX and Solaris SPARC machines respectively. The TOSSIM (Levis 03) and EMTOS (Girod 04b) emulators, both emulating TinyOS (Hill 00) applications, implement the traditional software-level emulation by intercepting and redirecting function calls when the program is being compiled. In this case, TinyOS system calls are redirected to a simulated TinyOS operating

system that provides all of the required functionality of the actual operating system but that can interact with the emulator. Using this approach, there is no need to intercept functions during the execution of the program.

It should be noted that some of the related works examined in this chapter use the term *simulation* when the term *emulation* would strictly be more accurate. TOSSIM, a TinyOS emulator is an example of such an instance, and these instances are noted throughout the thesis when they occur.

2.2.3 Alternative Software-Based Methodologies

There are several software-based approaches that could be adopted here in addressing these challenges but these are inadequate in meeting the broad challenges identified in chapter 1. For example, one approach would be to build a "super-simulator", an all-in-one simulator of the entire pervasive computing area. However, in order to build this type of simulator, it would require a level of domain expertise in all of the sub-domains in order to implement models of all the required elements and would not necessarily support areas of pervasive computing that are yet to emerge. Simulators that claim to be general pervasive computing simulators such as UbiWise (John J. Barton 03) commonly only model a subset of the complete set of aspects of the domain. Any approach that does not expressly incorporate flexibility and extensibility into its initial design and architecture is not well positioned to be adapted to the challenges of modelling overlapping and emerging new domains within pervasive computing.

A second alternative approach is the modification and integration of existing simulators. For example, researchers in academia and the motor industry have long speculated on the impact that the emerging area of vehicular ad-hoc networks, might have on vehicular applications including driver safety and vehicle coordination (Salim 08). There are many existing vehicle and network simulators so an integration of one of each of these simulator types should be able to leverage the independent domain expertise of both the vehicle simulation community and the network simulation community, and could address the modelling needs of the vehicular ad-hoc networking (VANET) community. Depending on the implementation and underlying formalism of the respective simulators, this approach requires an understanding

of common properties such as the simulated time and the location of common objects. In a simulated VANET scenario (Mangharam 06), the common objects, the vehicles and their network interfaces have the same location which must be synchronised within the respective vehicle and network simulators.

Clearly this approach can address the challenge of modelling overlapping domains, but until standardised API's are agreed upon for all simulators, this can only be achieved on a bespoke basis. At present there are no agreed interfaces, and few simulators that can inter-operate. The integration of two or more simulators does not address the challenge of being able to model new and emerging areas of pervasive computing, and this approach is only as extensible as its constituent simulators are. Finally, an integrated approach necessitates the tester to have the required expertise in each of the simulators that are to be integrated.

In conclusion, building a *super-simulator* is not a viable approach as it is inherently constrained with respect to future extensibility whilst efforts to integrate independently created simulators are limited by a lack of standardisation and by the flexibility and extensibility of the constituent simulators. So what are the alternatives? One approach is to incorporate flexibility, extensibility and re-usability from the start and make these principles core features of the adopted approach rather than add-ons at the end.

2.3 The Review Criteria

The simulators in this chapter are examined using eight review criteria, of which abridged working definitions are provided in table 2.1 and are subsequently expanded. These criteria can be categorised as follows. Review criteria one to three specifically map to the core challenges identified in chapter 1: namely flexible heterogeneity, extensibility and reusability. Review criteria four through eight address specific features that are common to pervasive computing simulators that merit examination and are included in order to draw out specific noteworthy features. These criteria can help to identify features of simulators that might not necessarily address the identified core challenges but may influence its architecture and implementation. As will be seen in the following definitions, there is some small overlap in some of the review criteria and this is highlighted when it occurs.

#	Review Criteria	Definition
RC1	Flexible Heterogeneity	Examines whether the simulator supports the flexible modelling of multiple instances of pervasive computing's common components.
RC2	Extensibility	Examines whether the simulator is extensible and can be modified with ease to support different pervasive computing domains.
RC3	Architecture (incl. Reusability)	An overview of the architecture from a software engineering perspective. Issues covered here may overlap with other evaluated criteria.
RC4	Application emulation	Examines whether the simulator supports the emulation of applications that exist within the modelled application domain.
RC5	Interaction between emulated and simulated components	Examines whether the interaction between instantiations of abstractions and emulated applications is supported.
RC6	Network simulation	Examines the support the simulator provides for network simulation of both wired and wireless networks.
RC7	Experimental support	Examines the support the simulator provides for evaluating simulations. This relates also to usability.
RC8	External Interaction	Examines whether the simulator is self-contained, or whether it is possible to interact with external simulators, services or processes.

Table 2.1: Simulator Review Criteria

Flexible Heterogeneity

This criteria examines whether the simulator supports the modelling of heterogeneous features of pervasive computing applications. These heterogeneous features may include a variety of sensors and other hardware devices, multiple applications possibly running on different platforms, users, the physical environment, and other features, both mobile and static. Additionally, the simulators are reviewed to determine their support for the flexible interaction of these heterogeneous features, a feature that is required if a simulator is to support emergent and cross-over pervasive computer application domains.

Extensibility

This review criteria examines whether the simulator is extensible by design. Systems are examined to determine what support exists for the modelling of new emerging application areas of pervasive computing not yet identified, where these application areas can comprise of both new hardware and software elements.

Architecture including Reusability

The third review examines the system's architecture from a software engineering perspective. Issues covered here may overlap with other evaluated criteria. The reusability aspects of the architecture are also examined in cases where the architecture can be used to model several pervasive computing domains.

Application Emulation

This review criteria examines what support the system provides for the emulation of applications that would typically exist within the modelled application domain.

Interaction between Simulated and Emulated Components

This review criteria examines what support is provided for modelling interaction between any simulated elements and any emulated applications within the one application domain. There is a small overlap here with the review criteria concerning flexible heterogeneity.

Network Simulation

This review criteria examines what support the system provides for the network simulation of any networked elements within the modelled domain. This may include both wired and wireless networks. This review criteria also examines what interfaces are available to algorithms and any emulated applications within the simulator.

Experimental support

This review criteria examines what support the system provides for the running, execution and subsequent evaluation of any modelled experiments. This could include for example scripting support, debugging features, or the ability to program user behaviour.

External Interaction

This review criteria examines the systems ability to interact with external simulators, services or processes. Often systems offer an API to connect external services for debugging or system extension. Any API's that are provided to support this functionality are examined.

Omitted Criteria

The simulators under review could also have evaluated against criteria such as performance, ease of use, and accuracy, however, the number of simulators attempting to address the generic modelling of pervasive computing is so few, that to benchmark these against each other would not be to compare like with like. The performance or "ease of use" of StarBED, a distributed platform for simulation can not be compared in a meaningful way against UbiWise, a simulator based upon a PC-based games engine.

The field of pervasive computing simulators is not as strongly developed as that of network simulation for example, where there are maybe twenty accepted simulators that can all be run on a local machine. In mature simulation fields such as these, the research questions have moved on from whether or not it is possible to model some aspect of network behaviour to how accurate and how quick that model is.

Finally, the accuracy and fidelity of any simulator is important, and this is captured within RC1 for simulated hardware devices, and in RC4 for emulated applications, and is a theme which is carried throughout the frameworks implementation and its evaluation.

2.4 An Overview of Pervasive Computing Simulators

Section 1.1 in chapter 1 introduced the many overlapping areas of pervasive computing, all of which involve different technologies and have different modelling requirements yet contain some commonalities. For example, smart spaces (Tapia 04) are often reactive applications, driven by sensors such as door contact switches embedded within the space, which can then adapt some aspect of the space to a user's requirements. An environment-monitoring wireless sensor network (Ji 04) could comprise of sensors, perhaps monitoring precipitation, attached to wireless nodes that propagate that sensor information to other nodes through an ad-hoc network. ITS systems (Klein 01) typically use embedded sensors such as inductive loops in the road network to improve the throughput of the vehicles in the system. All of these domains, and others considered, form the broad overlapping space that is pervasive computing.

This review of related work intersects several disciplines, as shown in figure 2.1 , some of which are subsets of pervasive computing and some that are independent research areas that have some commonalities with pervasive computing. There are many interpretations of pervasive computing, and occasionally simulators that are targeted at pervasive computing actually only address a subset of the area. Compared with an area of research such as wireless sensor networking, there are surprisingly few well-known simulators targeted specifically at the overall pervasive computing area. This is mainly due to its broad definition, and that few research groups address pervasive computing as a complete space, instead preferring to focus on specific pervasive application areas, such as context-aware or location-aware computing.

Simulators for pervasive computing can be broadly categorised into two groups. The first group includes simulators such as UbiWise (John J. Barton 03), Tatus (O'Neill 05), and the Lancaster simulator (Morla 04), which are pervasive computing simulators not addressing any particular sub-domain. UbiWise and Tatus support creating complex 3-D simulations of a pervasive computing environment such as a building or a smart-space, but not the components

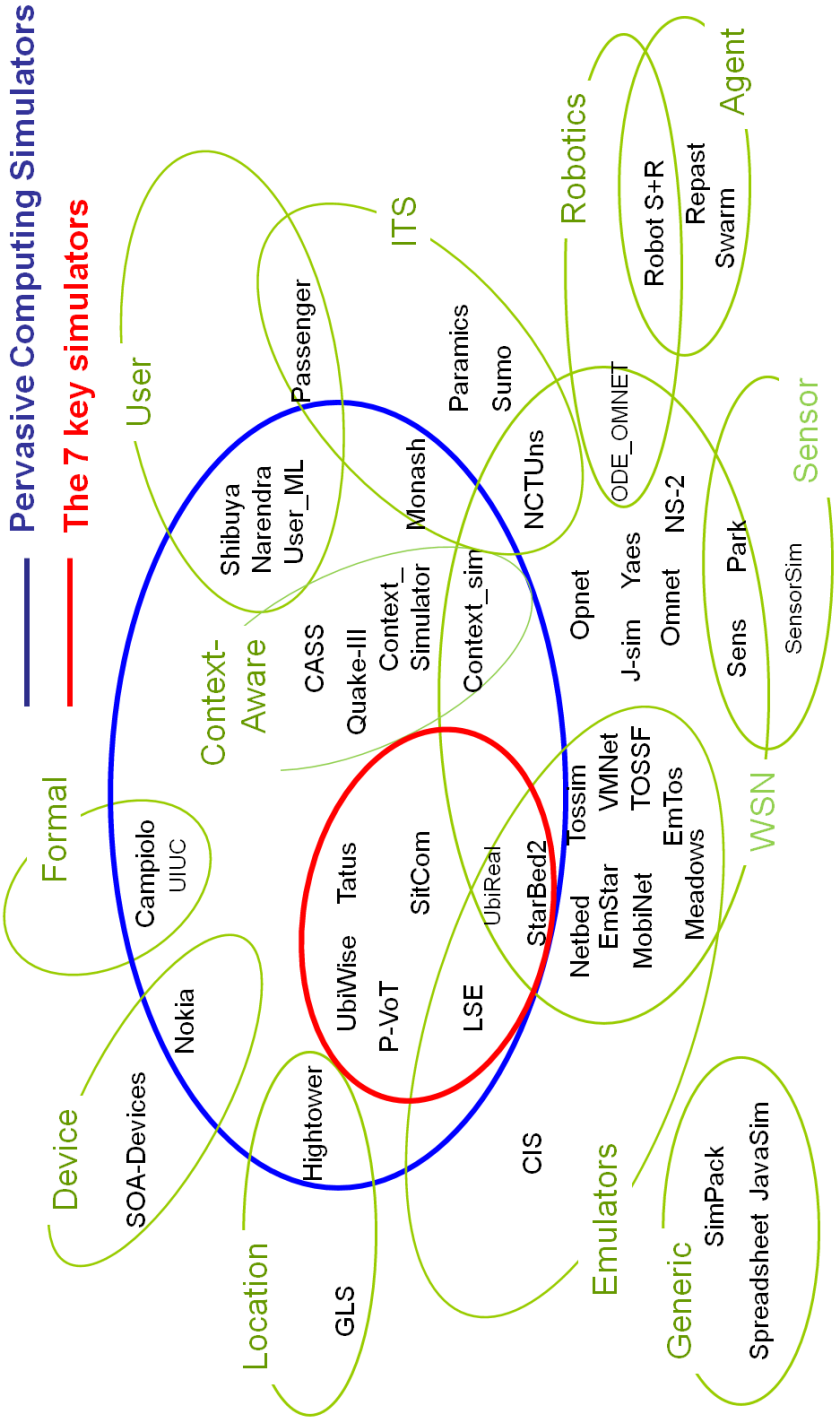

Figure 2.1: Overlapping Domains of Pervasive Computing Simulators

such as wireless sensors that may exist within those environments. The second group of simulators are those that address specialised subsets of the pervasive computing field. For example, Siafu (Martin 06a) is a simulator for context-aware computing applications. Additionally, there are many instances of simulators such as network simulators (Sundresh 04; Sobeih 05) and agent simulators (North 06; Narendra 05) being modified and used to model subsets of pervasive computing.

In fact, of the many simulators identified in figure 2.1, only those inside the red oval claim to specifically address pervasive computing, in the broad multi-disciplinary sense that this thesis considers. These seven are perhaps the most important of the simulators considered, as they attempt to address the challenges that have been identified for a flexible and extensible simulator for pervasive computing. An additional ten simulators, those inside the blue oval, and outside the red oval, either address subsets of pervasive computing, or are in the overlap of application domains such as intelligent transportation systems with pervasive computing.

To date, there has only been one simulator developed using a framework for the pervasive computing domain. Shibuya applied a framework approach (Shibuya 04) to modelling the spatial aspects of human behaviour in a pervasive computing environment. The focus of this work was on human interaction within these environments, rather than the environments themselves, however, and the physical aspects of the devices, and applications that might use those devices within a pervasive computing environment were neglected.

2.5 The Key Simulators

"..This thesis is concerned with methodologies that are both suitable for the evaluation *of a wide range of diverse pervasive computing scenarios* and that *can be extended to evaluate new scenarios* that may emerge in the future...". As seen in the previous section, there have been numerous works addressing a subset of the requirements of pervasive computing simulation, but few taking a more complete approach to modelling the domain as a whole. Of these simulators, seven are chosen that offer the most interesting and note-worthy approaches, and that are closest in spirit and execution to a framework-based approach to building flexible and extensible simulators

These seven simulators were chosen based on the following criteria. Importantly, they attempt to address almost all aspects of the identified review criteria, but with varying degrees of success.

Secondly, the typical user of the PiCSE framework is a researcher within a small to medium sized research group with limited access to hardware resources such as a cluster or even a cloud-infrastructure. Therefore, this review is concerned with lightweight simulators that are suitable for conducting experiments in a cost-effective manner. By lightweight it is meant that the simulator can execute on a single off-the-shelf desktop computer that does not require additional hardware to achieve a normal level of performance. As a by-product this avoids some of the downfalls of using distributed infrastructures which include the high costs, and having to negotiate and schedule shared access to that infrastructure.

Finally, this review focuses on simulators that are widely available and are being actively used within the pervasive computing community.

2.5.1 UbiWise

Ubiwise (John J. Barton 03) was one of the original simulators developed for pervasive computing, and was motivated by the need for rapid and cheap prototyping of pervasive devices and services. The simulator provides

> "a three-dimensional world, built on the Quake III Arena graphics engine, and serves to simulate a first person view of the physical environment of a user" (John J. Barton 03)

A simulated deployment enables the testing of pervasive services, implementations of protocols and integration of devices. Essentially, UbiWise is a human-in-the-loop simulator, which allows users to interact virtually with simulated devices in a 3-D space. The simulator aims to mix simulated and prototype devices and services where possible.

RC1: Flexible Heterogeneity

UbiWise's environmental modelling is provided by using proprietary Quake III formats, and editors such as GtkRadiant are available to develop interactive 3-D models. These editors provide a graphical interface for modelling the environment allowing a developer to position devices, walls, and so forth. The most basic object within UbiWise are 'devices' and these are specified through an XML device description file and associated Java class files. The graphical representation of the device is also encoded in the file, by specifying the shape of the object and associated JPEG images for various aspects of the device.

Supporting new sensors, "whether handheld or environmental" is one of the stated aims of the UbiWise platform and location trigger-based sensors, device inputs as well as networked sensors are all described theoretically. User models are an intrinsic part of any game based development platform and are available.

RC2: Extensibility

The solution proposed is extensible in that new devices, sensors and java based applications can be integrated into platform, however the platform itself is a device-centric and whilst readily suited to smart-space type environments, it would be difficult and time-consuming to extend the platform to model other types of environments.

RC3: Architecture

UbiWise uses a client/server architecture which is built upon the Quake III game engine. A single UbiSim server hosts each simulation and maintains a 3-D representation of the environment. A Wise[5] client, representing a user and a hardware device then connects to the UbiSim server. The client maintains a consistent view of the environment by exchanging physical environment messages with the server, which are then propagated to other users. The device view and the device's interaction with the environment are similarly handled by passing device interaction messages. The combined system of the UbiSim server and Wise clients is known as UbiWise.

[5]Wireless Infrastructure Simulation Environment

RC4: Application Emulation

Wise supports the execution of application code in the form of Java .class files and since the developers of the system describe simulating user inputs such as mouse events, then it appears that at least these device oriented applications are at least emulated.

RC5: Interaction of Emulated and Simulated Components

Applications within UbiWise are intended to interact with simulated devices and these devices can consume events that are driven by events within the simulated environment, such as location-based sensors. There is no evidence of interaction of emulated devices with actuators although this is most likely to be achievable through hard-coding.

RC6: Network Simulation

UbiWise supports the simulation of characteristics of network behaviour such as variable latency and intermittent connections but provide no details of how this simulation is achieved so this is most likely to be achieved with their own proprietary simulator.

Additionally, network interaction within this simulator can be achieved with an actual live network interface. As this is a human-in-the-loop simulator, this is possible as a simulation runs at the same speed as in the real world.

RC7: Experimental Support

Wise is a research-driven tool for rapid prototyping and provides excellent support for the running of experiments. Debugging, tracing of application behaviour as well as user-centric controls such as record and playback make it easy to evaluate applications with the Wise system. Beyond evaluating applications, there are no described mechanisms for the same level of tracing although a record and playback function is likely to be provided within the graphical part of the platform.

RC8: External Interaction

Part of the Wise device specification, modelled in XML, allows the application's .class files to build upon protocols code such as HTTP, enabling the integration with external services outside the control of the simulator. As UbiWise runs in wall-clock time[6], protocol issues such as latencies in the network do not have to be simulated but are introduced naturally as the simulated device exchanges HTTP messages with external HTTP services.

Summary

In UbiWise, the developers correctly claim to have achieved a balance between "fidelity" and "simplicity". Fidelity in this case meaning fidelity to the user interaction experience, and to a realistic environment. In addition to this, UbiWise offers realistic modelling of application-level devices and an interactive 3-D model of the environment in which they exist. Enabling simulated devices to interact with external services lends even more credibility to this claim, however overall, UbiWise is deficient in several aspects. It does not provide a flexible architecture for simulating anything other than hardware devices, and their embedded functionality, and the environment in which those devices exist. Hardware devices, such as sensors and actuators, which are common to pervasive computing scenarios, can be modelled but no framework or generic abstractions are provided to support the development of these models. They must be built from scratch using the Quake native Software Development Kit (SDK).

Furthermore, no metrics are provided regarding the performance or the accuracy of the simulator. As UbiWise only attempts to prototype application devices and their environment, any analysis regarding accuracy and performance are valid only in relation to the modelling of the image representation of the devices and the environment. In this respect, UbiWise is arguably effective. The modelling of devices is both accurate and realistic from the perspectives of both the look-and-feel and the functionality of those devices. However, this modelling comes at the expense of the time and effort required to develop these models. The developers

[6]Wall-clock time is when one second of simulated time takes one second of actual time to execute. This is required as UbiWise is a human-in-the-loop simulator.

themselves acknowledge the large effort required to build the graphical models required for each simulation.

Finally, UbiWise is not suitable as an experimental platform. Support for running sets of experiments does not seem to be built into the design and an API exposing data of interest for experiments has not been provided. Furthermore, the human-in-the-loop model employed does not lend itself to evaluating scenarios or running large sets of repeatable experiments. Introducing a random element such as human behaviour makes it difficult to replicate experiments thus rendering the platform unsuitable for experiments other than those in which human interaction is a requirement. This is, however, one of the main selling-points of UbiWise. The lengthy development period is an additional and potentially time consuming factor in running large sets of experiments with varying parameters.

UbiWise is primarily a graphical simulator however, and not an experimental tool that is easily extended to new scenarios. As such, it is not surprising that it is deficient in meeting many of the requirements identified for this thesis. It is, however, one of the original and few simulators targeted specifically at this domain, and therefore it is included.

2.5.2 Tatus

Tatus (O'Neill 05) adopts a similar approach to UbiWise to modelling pervasive computing systems. A well-known game engine, Half-Life, is adopted to simulate immersive pervasive computing environments in which users and applications can interact with virtual worlds.

RC1: Flexible Heterogeneity

Using Half-Life's graphical editors, it is possible to create complex 3-D implementations of physical environments, such as buildings, rooms and open spaces. A proprietary format, Binary Space Partitioning (BSP), is used to capture the topology, shape, and even texture of the environment as a binary tree which ensures that the environment can be both mapped and explored efficiently. In addition to this, users and their actions can be accurately portrayed by using either a) a real human to drive the simulation or b) programming a "bot", or Non Player Character (NPC), to replicate these actions. Sensors and actuators are simulated by means

of an API allowing applications to query and to act upon the simulated environment. The developers note that only simple actuation is possible due to method parameter restrictions: only a single unparameterised use() method is provided.

RC2: Extensibility

Similar to UbiWise, the Tatus platform provides extensibility through the games engine upon which it is built. The Half-Life SDK allows the creation of new physical spaces and for sensors that can measure data within that environment. Although the game's SDK does provide a means of extensibility, the developers of Tatus note the difficulty and steep learning curve in using that, highlighting that they do not feel that the SDK is an appropriate means of providing extensibility for resource-constrained researchers.

RC3: Architecture

The Tatus architecture is composed of three main components. The simulator, a modified game engine is the main component, which takes a map definition file and an XML file defining the message format for the sensing and actuation API. A Java proxy communicates a network with the games engine, and provides an API for sending and receiving messages to the simulator. Any applications that are "under test" can use the Java API to interact with the simulator.

RC4: Application Emulation

The external proxy provided by the Tatus architecture does not provide an emulated interface for applications that are connecting to the simulator. These applications must be built upon the proxy's API.

RC5: Interaction of Emulated and Simulated Components

There are no emulated software components within the Tatus architecture, and hence there is no interaction between emulated and simulated components. Un-emulated applications

that are connected to the simulator via the proxy may interact via the sense and query APIs provided.

RC6: Network Simulation

An architecture has been proposed for the integration of a third party network simulation tool, TOSSIM, into the Tatus simulator for the integrated simulation of wireless networking scenarios in pervasive computing environments. The proposed architecture creating a new interface between the network simulator and a "Real life Simulator", itself a proxy that can interface with the Tatus Simulator. The location of wireless nodes, and their mobility patterns would be specified by the Real Life Simulator.

RC7: Experimental Support

The simulator also supports the running of experiments through the inclusion of three features, that exploit the underlying Half-Life infrastructure. Multi-player games allow experiments to be run with up to eight applications and eight users, a constraint imposed by the underlying game engine. As previously mentioned, Tatus allows reproducible experiments to also be implemented by removing the human element and replacing it with an NPC. In addition to this, experiments can be saved, re-run and logged, using the Half-Life game API, allowing off-line analysis and evaluation to be performed.

RC8: External Interaction

Tatus employs an external proxy that enables the integration of applications, termed Software Under Test, into the simulated world. The proxy, which is not necessarily running on the same machine as the simulator, provides an API allowing applications to both query and update the world. Events of interest are pushed to a database in the proxy, whereas the proxy's query API allows applications to "extract" real information from that database. Similarly, actuation events are performed using an actuation API, that pushes events directly into the simulated environment. XML messages are used to exchange data between the simulator and the proxy which then pushes the data to the application, and this exchange is also supported remotely

across a network.

Summary

The aim of Tatus is to support the incorporation of real-user behaviour into a simulation. In this sense, similar to UbiWise, it can be argued that Tatus allows the accurate modelling of both users and physical environments. These are however only two aspects of pervasive computing. Both sensors and actuators are abstracted to an API, provided by the proxy, mediating interaction with Tatus's model of the environment. Furthermore, the simulator's ability to be extended to actually include different categories of these devices is supported only by the Half-Life SDK. There are no generic models of devices, although developers can utilise Half-Life's support for event triggers to generate events when certain criteria, such as proximity, are met.

The proxy-style support for external applications works quite successfully, and allows many applications to interact with the simulator. Ultimately however, these applications have to be built upon the proxy's API, and there is no support for emulating the inputs and outputs of the data streams between the proxy and the application.

Finally, there is no support for integrating a network simulator into this work, although it has been identified as future work and a potential architecture has been proposed. However, since Tatus runs with a human-in-the-loop, applications can interact outside the control of the simulator with networked services if required.

2.5.3 Lancaster Simulation Environment

The Lancaster hybrid test and simulation environment (LSE) (Morla 04) is an environment that supports the integration of third party simulators to evaluate location-based applications. Support is provided for external applications, using a Web Services interface, allowing them to interact with simulated environments. The control and interaction of both, simulators and applications, are mediated and controlled by a Systems Manager that is also responsible for overall experimental control.

RC1: Flexible Heterogeneity

The LSE simulator is built using an open approach allowing the connection of third-party simulators that can simulate different aspects of pervasive computing environments such as mobility or context simulators. No description is provided of how these simulators might be extended in a manner defined by the LSE architecture and any integration may only be achievable if the third party simulators are designed to provide this functionality.

RC2: Extensibility

LSE's approach of supporting the integration and mediation of third-party simulators is inherently extensible. In separating other features such as the experimental support and the application daemons, the simulator has provided a framework for potentially targeting new and emerging pervasive computing domains.

RC3: Architecture

LSE is built upon a distributed architecture which mediates the control and execution of applications and third party simulators. A system manager controls the execution of simulators and applications using a Web Services interface allowing these to be executed on separate machines. Low-level kernel access is required for the machine on which applications are being tested. The System Manager is also responsible for gathering experimental results and provides data to a graphical user interface.

RC4: Application Emulation

Supporting actual application code is one of the requirements of the LSE test environment, and the simulator distinguishes between two aspects of the emulation required for those applications so that the minimum number of changes to the application code are required.

The first aspect is that the emulation component must support a realistic network interface. Applications should be able to send and receive messages across both 802.11 wireless Lan and GPRS. The emulation component should additionally provide interfaces for

receiving any location information and any kind of context information that might be relevant for the application behaviour.

RC5: Interaction of Emulated and Simulated Components

The LSE architecture enables the integration of the emulated applications with third-party integrated simulators. By providing a "context" api for example, an emulated application can interact with a context simulator over a Web Services interface.

RC6: Network Simulation

LSE integrates a popular network simulator, ns-2 (Fall 01) into their test environment, allowing the simulator to model the wireless communication between applications. This is done without modification of the application code. Using this software emulation technique, packets generated by the applications are intercepted in a modified kernel and redirected through the ns-2 simulator. Importantly, this approach is transparent to the applications themselves.

RC7: Experimental Support

LSE provides good support for experimental control. Application daemons are provided enabling instantiations and monitoring of individual applications. Application outputs and error streams are recorded by a system manager for logging which also can control and monitor third-party simulators such as the network and location simulators.

RC8: External Interaction

LSE executes in wall-clock time and interaction with external services is both possible and transparent in simulated scenarios. This is achieved using the web services based Global Grid Forum[7] standards to "disseminate the sensor data to other Grid applications", that themselves contribute to the overall simulated scenario.

[7] http://www.globus.org

Summary

No abstractions are provided for the classical pervasive computing components such as sensors or other hardware devices. However as noted, LSE's flexibility or suitability as a pervasive computing simulator is achieved through the use of a Web Services interface which allows third-party simulators to be attached to the overall system. As will be shown later in this chapter, there are simulators that address individual aspects of the overall pervasive computing space, and potentially these could be integrated into the LSE simulation environment. Using Web Services to integrate third party simulators offers potential extensibility, however, some concerns arise. The integration of multiple simulators acknowledges and accommodates the expertise that each brings. These simulators can provide complimentary modelling of separate aspects of pervasive computing scenarios, yet there is no concrete architecture within LSE, to support the controlled interaction of multiple simulators. For example, there is no specification of a common user model, that might allow the user model from one simulator, to interact with a sensor model from a second simulator.

2.5.4 UbiREAL

UbiREAL (Nishikawa 06) is a smart-space simulator, that extends the traditional "modified game simulator" employed by UbiWise and Tatus to offer a more complete simulation environment. In addition to graphical 3-D environment models, the UbiREAL simulator allows users to specify physical quantity simulators and also to integrate simulated and real applications in the same virtual smart-space. The basic UbiREAL architecture comprises four components that interlink to achieve these goals. These components are: physical quantity simulators, a network simulator, a visualiser and GUI, and application programs.

RC1: Flexible Heterogeneity

The UbiREAL simulator supports the integration of physical quantity simulators. These simulate

> "invisible physical quantities such as temperature, humidity, electricity and radio

> as well as visible (audible) quantities such as acoustic volume and illumination." (Nishikawa 06)

A developer must specify a physical quantity in terms of an appropriate physics formula, for example, specifying the rate at which heat dissipates given temperature differences in different areas. This model is supported by a publish-subscribe event mechanism, which enables the interaction between sensors and the physical quantity being sensed. Very simply, as the physical quantity changes over time, sensors, which are implemented as software drivers, are notified of these changes, and a sensing event can occur.

RC2: Extensibility

The UbiREAL structure has been designed as a closed and complete system. Integration of third-party simulators and emulation libraries is not considered, however a framework for the simulation of physical quantities does provide a means for extensibility with respect to environmental, sensing and potentially actuating models. A methodology for integration of emulated applications has been proposed for two emulation API's and conceptually, it is feasible to consider how this approach may be extended for other application libraries that the UbiREAL simulator may integrate in future work.

RC3: Architecture

UbiREAL is a modular component-based architecture comprised of four interacting modules. At the centre of the architecture is a smart-space designer and visualiser which defines the physical environment, the components within and a means for executing the simulation. This module takes provides inputs to the network simulator, and also provides a physical environment in which emulated applications can be executed. Finally, the central module also provides an environment and informs the final module, which is the simulator for physical quantities.

RC4: Application Emulation

UbiREAL allows the emulation of applications built upon common libraries. The emulation is achieved after by linking with libraries that interface with UbiREAL, instead of the original libraries that the applications were built upon. This linking is performed between the compilation and execution of the program. Thus, any application invocations of that library are redirected to the simulator, achieving emulation. The BSD Socket and java.net libraries, are both supported, and intercepted calls to these libraries are redirected to UbiREAL's bespoke network simulator. Emulation of applications built on software device drivers is achieved in a similar fashion. Calls to the modified device drivers retrieve values from the physical quantity simulators, and return them to the application.

RC5: Interaction of Emulated and Simulated Components

No direct interactions are defined between emulated applications and simulated physical quantities. Direct readings from the physical quantities may be obtained through "device drivers" but this is not provided through an emulated API.

RC6: Network Simulation

UbiREAL provides its own internal network simulator for the simulation of wireless communication. It supports models including the free space model, the line-of-sight model and the ray-tracing model, and their model takes into consideration variables such as the transmission range, received signal power and wavelength. Radio propagation is calculated before the simulation is executed, and in simulations involving mobile nodes, the radio propagation is calculated periodically. Periodic calculations may result in some loss of simulation accuracy with respect to the wireless model.

RC7: Experimental Support

Finally, UbiREAL offers a facility for the systematic testing of smart-space applications. It uses a formal methods approach to modelling the domain, in terms of sets of smart-spaces U, rooms R, and devices D. The approach also models physical quantities as *attributes* in each

room. A service specification is defined as a set of rules AP, and a set of propositions, P. Using this formal approach, an experiment can be described as a service specification *Spec=(AP,P)*, that specifies a range of values that *attributes* can vary over, across many rooms, R, comprising a smart space, U. As these attributes change during the run of the experiment, events are generated and pushed to applications, thus exercising them.

RC8: External Interaction

Using a similar approach to LSE, a redirection of calls is performed at the networking layer within the operating system, to enable the integration of real applications with virtual applications. A modified NAT[8] process on the host computer of the real application examines all network packets, and packets destined for emulated applications are modified and redirected to the address of the host running the simulator. This occurs at the TCP level with the result that no modifications are required to enable external applications to interact with real applications.

Summary

The UbiREAL simulator integrates many novel and practical features into its architecture. Particularly, the integration of real and virtual applications is an enhancement of LSE's similar approach to integrating external applications. Similarly, it extends the graphical approach adopted by UbiWise and Tatus, to include a model for simulating physical quantities, providing a more complete model of an environment that is "measurable" by sensors.

Although the UbiREAL simulator runs in wall-clock time, an approach that can lead to lengthy experimental times, the inclusion of a formal approach to modelling the experiments is a significant advancement in the state of the art. Its integration into the simulator makes up partially for the inherent shortcomings of using a graphical game-based approach to run experiments.

However, the UbiREAL simulator is constrained to smart spaces and modelling environments in terms of rooms, and spaces only. It is difficult to see how this approach could be

[8] Network Address Translation

extended or modified to include large-scale scenarios, or scenarios where the devices or important components are anything other than applications or sensors. The formal approach to modelling is constrained to smart-space scenarios, and experiments run within those scenarios, although potentially the modelling language used could be extended to support a broader range of scenarios.

2.5.5 SitCom

SitCom (Fleury 07) is an open and extensible framework developed at IBM that assists in

> "developing context-aware services, editing and deploying context situation models, and simulating perceptual components." (Fleury 07)

SitCom which stands for Situation Composer, provides abstractions of the main components in context-aware computing services, and provides a two-tiered architecture that supports the creation of models of these scenarios. SitCom itself provides the underlying framework, providing functionality such as modelling the environment, in both 2-D and 3-D, and also an array of managers for managing the inputs and outputs of a simulation instance. SitMod is an instantiation of the framework abstractions that forms a situational model, e.g. an instance of a particular simulation or experiment.

RC1: Flexible Heterogeneity

The working definition of *context* that SitCom uses is that of Dey in the development of the Context Toolkit (Dey 00). Here context is defined as

> "any information that can be utilised by an application, including sensed and synthesised knowledge on users, the objects of the scene, situations, the application itself, environment, and world." (Dey 00)

SitCom provides a set of abstractions based around this definition that allow context-aware computing scenarios to be modelled. These abstractions are partitioned into five categories, each addressing a general class of abstractions. *Entities* are defined as features that can

be mapped one-to-one to real world objects. *Entities* have a set of attributes, that can be updated during a simulation, and can be the inputs for *Sensors*. *Sensors* consume attributes of particular entities, and produce raw sensor data. *Perceptual components* consume data streams from sensors and provide the mapping from sensor data to higher level contextual data. Contextual data, modelled as facts, is consumed by instantiations of *Situation Modelling* abstractions, and provide information about situations, which consists of a set of *states* and their associated transitions. Finally, a *Services* abstraction is provided allowing context-aware services to be developed using application logic, and user interface design.

RC2: Extensibility

The SitCom simulator does not consider extensibility as a key requirement. It is targeted specifically at the context-aware computing sub-domain of pervasive computing and its architecture, described in the following section, reflects that. The simulator does not support the integration of third-party simulators that might assist in achieving that extensibility.

RC3: Architecture

SitCom is described as a four-tier architecture. At the lowest level, sensors provide raw sensor data, which is consumed by perceptual components which are at the second level. Perceptual components add a level of contextual or semantic meaning to that raw sensor data. Situational modelling is the third tier and here information from the perceptual components is used in an application specific manner. For example, an application here could interpret a range of sensor inputs including presence and activity to detect whether a room is in use or not. The top tier of components are service-oriented components which act upon individual applications such as the presence application. The SitCom simulator is implemented as an extensible Java toolkit that provides functionality that support the development of simulations at all four tiers of its architecture.

RC4: Application Emulation

Application emulation is not one of the requirements for the SitCom architecture and applications are composed as part of the Java runtime within the simulation.

RC5: Interaction of Emulated and Simulated Components

As there is no emulated applications within the SitCom simulator, no interaction between those applications and simulated components is noted. However, application classes that are deployed can consume both simulated and real contextual information from perceptual components.

RC6: Network Simulation

Network simulation is not incorporated as a module within the SitCom architecture. The SitCom framework targets context-driven application and considers standalone applications that consume raw sensor and contextual information as their inputs. It can be inferred that some network component may exist within applications deployed at the services tier, however these are likely to interact across real networks and not through controlled simulated networks.

RC7: Experimental Support

The SitCom simulator allows experiments to be specified through its GUI, allowing predefined situations (experiments) to be loaded. "Synthetic" or "recorded data" can then be fed into the simulation. In particular, SitCom supports the loading of semi-simulated, and real data into a simulation, allowing the higher-level abstractions such as the *Perceptual Components* and the *Situation Modelling* abstractions to exploit these real data sources.

RC8: External Interaction

A remote API is provided by the SitCom simulator allowing standalone modules to interact with the simulator. These remote modules are part of the situational modelling tier of SitCom components and the API therefore provides methods from querying simulated perceptual components.

Summary

SitCom provides a set of abstractions that model a range of aspects of context-aware computing. What is unusual and relatively novel about these abstractions is that they are aligned in the vertical domain, allowing the modelling of all parts of a context aware application. In addition to this, it enables the modelling of complex sensors such as video cameras, and higher-level perceptual components such as a Body Tracker that can exploit these complex sensors. SitCom has been used to model a range of context-aware applications, such as a Meeting State Detector and Crowd Detector amongst others, and these scenarios leverage the range of abstractions provided by the simulator.

SitCom's ability to integrate data streams from external sensor sources is a novel feature in this domain, and allows the evaluation of context-aware applications against real data sources, enabling a stronger validation of those applications.

The authors express that flexibility is achieved, and this is true for the context-aware computing domain. However, it is difficult to see that SitCom might be flexible enough to extend to modelling other subsets of pervasive computing. Perhaps the most glaring omission from the specification of SitCom is that network simulation is not supported internally or externally.

2.5.6 P-VoT

P-VoT, the POSTECH Virtual reality system development tool, has been extended by (Seo 05) to provide a virtual pervasive computing environment, that is suitable for rapid prototyping. Specifically, the work is targeted at a subset of pervasive computing where the accurate modelling of the display of data to users is the most important requirement. Although this thesis and state of the art is not concerned with multimedia displays in the general case, the implementation of the simulator offers some novel features, and it is therefore considered. These features include the ability to model the behaviour of devices in a Statechart and the use of a scripting language to associate functionality with that Statechart.

RC1: Flexible Heterogeneity

The simulator implements three classes of objects, that reflect the targeted multimedia domain. These objects are sensors, displays, and processing objects. An instantiation of the P-VoT simulator is a composition of these objects, which can interact in a pre-determined fashion. Sensors push data to processors, which then control the output of the display. A display is analagous to an actuator in this particular pervasive computing domain.

Each component can be specified at different levels of abstraction offering an additional level of flexibility. For example, sensors can be instantiated to produce raw data, filtered data, or even higher-level data such as contextualised data. Processor objects can be instantiated to implement raw data processing patterns such as filtering, or higher-level processing such as pattern recognition. Finally, displays can be specified at the level of LEDs[9] or text at the most basic level, up to 2D and 3D displays.

The interaction between these objects are modelled using Statecharts. A Statechart specifies the states, the transitions between states and the messages that can be sent between instances. Upon reaching a certain state, some Python (a scripting language) code is invoked. The interactions between the different object types is captured within the Statechart. For example, a model of a passive sensor, remains in an *idle* state until an object or the measured phenomenon meets some criteria and the sensor undergoes a state transition. This change in state can result in the invocation of a Python method, whereby further actions and computations can be carried out.

RC2: Extensibility

P-VoT's scripting based approach is low-level but potentially extensible. One of the advantages of scripting languages, that they allow rapid prototyping, has been capitalised upon by the developers and it would be feasible for script-based extensions to be built within the system specifying new application behaviour and new simulated models. Indeed the developers of P-VoT have validated their simulator's approach against other pervasive computing domains beyond context-aware applications, which include smart devices, smart environmental

[9] Light Emitting Diode's

monitoring, and smart spaces such as an information kiosk.

RC3: Architecture

The P-VoT simulator is a component based simulator that is comprised of a set of authoring tools. An integrated development environment (IDE) provides a Statechart editor, a library of reusable virtual and interaction objects and a Python execution environment. The combination of tools runs on a local machine, and provides a graphical user interface for the creation and specification of simulated experiments. No external interaction with external services, simulators or external applications is supported within the architecture.

RC4: Application Emulation

Applications are not emulated within the P-VoT system. Instead, they are modelled as "processing objects" and are implemented using Statecharts and the Python scripting language. Within these Statecharts, each state is defined by a script containing algorithms defining behaviour and outputs based on existing states and sensed data.

RC5: Interaction of Emulated and Simulated Components

As there are no emulated applications in P-VoT, there is no interaction between emulated and simulated components. Applications that are modelled as state-charts and behaviours may access simulated sensor information and use this as part of the scripted application behaviour.

RC6: Network Simulation

There is no network simulation component within P-VoT. The focus of the simulator is on context-driven, multimedia based pervasive computing applications, a very niche domain within pervasive computing that does not necessarily require network simulation.

RC7: Experimental Support

The P-VoT simulator provides experimental support via an artifact described as a data collection objects. These can collect application performance data as well as aggregate data

provided by users of the simulator such as usability information and survey answers.

RC8: External Interaction

No external interaction is supported within the P-VoT simulator.

Summary

Like SitCom, P-VoT is a simulator targeted at a very particular subset of pervasive computing, in this case context-aware multimedia enhanced pervasive computing environments. With this in mind, it is perhaps unfair to judge the simulator on the criteria by which other simulators have been judged. However, this simulator has adopted an approach to specifying the behaviour of objects that is novel in the area of pervasive computing simulators. The use of a combination of a Statechart and the Python scripting language results in a rapid prototyping and evaluation model. The result is a simulation that can be rapidly prototyped and implemented. The developers report that the overall authoring process, "simple scripting, creating states, and transitions", can take less than an hour, for a user with experience of the system.

2.5.7 StarBED2

The last of the core pervasive computing simulators to be considered is StarBED2 (Nakata 07), which is a large-scale hybrid of both simulator and physical testbed for pervasive networks. Although not a classical simulator in the sense of what is considered for this thesis, it is nevertheless worth considering. Specifically, StarBED2 enables the emulation of hundreds of thousands of heterogeneous nodes within pervasive computing networks. In order to do this, it outlines the required functionality to exercise what it has determined to be the most important characteristics of pervasive computing networks. These characteristics are that:

- A realistic pervasive computing scenario comprises heterogeneous sensor nodes.
- Those nodes may have a high spatial density.

- The interaction between the environment and those nodes is an important factor in the overall model, which is a characteristic of pervasive computing.
- The location of those nodes is important in modelling the interaction of these nodes and their environment.

In order to achieve this, StarBED2 provides a physical testbed that emulates the physical environment, supports the emulation of several node types and architectures, and supports the interaction of the environment and those nodes. The overall architecture of the distributed StarBED2 testbed is not considered here, only the architecture at a single node within that testbed.

RC1: Flexible Heterogeneity

The environmental space with StarBED2 is achieved through either statistical models, or through a "realistic manner" where the physical phenomenon is simulated according to some domain expertise. There is no support within StarBED for the emulated concept of a sensor, rather that the environmental space is a data source which can provide inputs to nodes that are within that modelled space.

RC2: Extensibility

The fixed and hardware centric approach that StarBED2 has used is not conducive to extensibility to other pervasive application domains. The focus of the testbed is on the realistic testing of physical nodes in fixed locations and accounting for mobile nodes such as physical users or vehicles would be difficult within the physical testbed described.

RC3: Architecture

The StarBED2 architecture is logically structured as a three tier architecture. At the lowest tier, the network space provides the communication infrastructure of the physical testbed, which is composed of approximately seven hundred personal computers (pc). Two networks connect the nodes. The experimental network is the network in which experiments run and

communicate. The second network is the management network, a separate network allowing a user of the testbed to remotely manage the experimental setup and the configuration of the nodes within that experiment. The physical testbed can also be logically partitioned and can therefore allow multiple experiments to run with difference parts of the physical testbed.

The second tier of the architecture is the node tier, in which emulated applications run and have access to the underlying physical testbed. Finally, the top tier of the architecture is the environmental space, a simulated environment which nodes can interact with, either querying or acting upon that environmental space.

RC4: Application Emulation

Emulation of heterogeneous nodes within StarBED2 is accommodated at three levels: The middleware level, the system call level and the instruction level. Depending on the targeted architecture and the implementation of the architecture, the appropriate level is chosen. For example, emulation would be performed at the middleware or system level when the application on the node is specified in a high-level language. Alternatively, emulation would be performed at the instruction level if the operating system on that node was not fully available, and if applications were being written close to the hardware layer. In the cases of system call and middleware emulation layers, the emulated application must be recompiled and linked against a library that redirects to the StarBED2 testbed instead of the physical hardware. In the case of instruction-level emulation, an instruction translation is performed to map instructions to methods invoking the corresponding methods within StarBED2.

Each emulated node then runs within its own VMware[10] virtual machine, making each node virtual. Up to ten nodes are then run on each physical machine within the testbed, and the responsibility then transfers to the VMware software to manage the execution of the individual virtual nodes.

[10] http://www.vmware.com

RC5: Interaction of Emulated and Simulated Components

The interaction between emulated applications and their environment is captured as a series of conceptual spaces and conduits between these spaces. Three types of spaces exist, an environment space, a network space, and a node space. Within the node space, the physical location of the device is critical and it affects the nodes interaction with the environment space, also addressed through physical coordinates. The node's location within the network space consists of an IP address and port number pair. The logical structure of StarBED2, as a series of spaces and conduits manage the bindings between the physical coordinates and the network coordinates, enabling the interaction between nodes and their environment, and amongst nodes themselves.

RC6: Network Simulation

The StarBED2 testbed is a physical testbed and the underlying hardware components are networked using wired and wireless components. There is no network simulation component within the testbed as the goal of the testbed is to move beyond simulation into hardware based testing.

RC7: Experimental Support

A separate management layer within the physical part of the testbed is an indicator of how experimental support is managed. A controller tier manages the execution and configuration of individual nodes within the testbed and can record logged information out-of-band, i.e. the management overlay of the testbed does not affect any ongoing experiments, a potential problem in distributed testing architectures.

RC8: External Interaction

Emulated nodes within the StarBED2 architecture run in a real-time environment and on their virtual machine within a physical node. As there are no constraints on the application code that run on the emulated nodes, it can be inferred that these can interact with external sources without affecting other nodes within the overall. experiment.

Summary

StarBED's approach to emulation is flexible and relatively unique within the pervasive computing space in that it enables the emulation of heterogeneous nodes. The implementation using a VMware solution eases this task to a degree, in that it transfers the responsibility of managing the individual instances to the VMware server. This partially addresses one of the main problems of emulating multiple applications: application synchronisation. As applications execute in wall-clock time, it is possible for inaccuracies to enter the simulation when applications do not run in their correct ordering, a problem, that is exacerbated when trying to integrate emulated real devices with simulated models. StarBED2's VMware-based approach addresses this problem, however the overall solution is an expensive one. Running a VMware server is costly in terms of machine overhead, a fact noted by the developers of StarBED2 and one of the reasons why the number of nodes per single machine within the testbed is restricted to ten.

The developers make no effort to enforce causal ordering with respect to the ordering of individual application executions within a VMWare instance. In restricting the number of nodes per instance to ten, they attempt to mitigate the negative side affects of overloading the VMWare. The developers note, that on the original StarBED testbed, upon which StarBED2 is based, that when above ten virtual nodes were run, then "realism of experimental results becomes unacceptably low".

2.6 Perspectives

As a result of the wide range of simulator domains examined, cases will arise where a simulator may only address a subset of the review criteria. For example, a pervasive computing simulator may provide support for environmental models but not for actuators that interact with those environments. In these cases, the simulators ability to meet a criteria is marked as not-applicable, N/A. In general, however, the simulators are examined against all of the applicable review criteria, and are marked as either partially, ○, or completely, ●, meeting that review criteria. Review criteria 3, the system's architecture is omitted as it is too subjective to

	Flexible Heterogeneity	Extensibility	Application Emulation	Component Interaction	Network Simulation	Experimental Support	External Interaction
UbiWise	○	○	○	●	n/a	n/a	●
Tatus	○	○	●	●	○	○	●
LSE	○	●	●	●	○	○	●
UbiREAL	○	○	●	●	●	●	●
SitCom	○	○	○	○	○	○	○
P-Vot	○	○	n/a	n/a	●	●	n/a
StarBed2	○	○	●	●	●	●	●

Table 2.2: Comparing the review criteria for the key pervasive computing simulators

compact the review in a single column.

The evaluation criteria for the simulators discussed have been met with varying degrees of success. Table 2.2 reveals that LSE, UbiREAL and StarBED2 were the most successful in meeting these requirements, although none met all of them completely. Two important questions are raised. Are all of the requirements necessary? Why are some of the requirements not met? To answer the first question, one must examine table 2.2 again. What can be observed is that none of the original six requirements are completely neglected. Only requirement four, support for network simulation, is not addressed by more than one simulator. In this case, two simulators, SitCom and P-Vot were targeted at sub-domains of pervasive computing: context-aware computing and multimedia enhanced smart-spaces. In these cases, an accurate model of the user and the environment from its perspective is more important than the modelling of the interaction of applications within that domain. In general though, all of the simulators examined address the requirements to some degree, but do not necessarily meet them all successfully.

To answer the second question, the answer can again be found in SitCom and P-VoT. These are two simulators that target specific domains within pervasive computing, domains that do not require network simulation. The requirements that have been identified though, are that

Component Type	Environment	Sensor	Actuator	User
UbiWise	●	●	n/a	●
Tatus	●	○	○	●
LSE	○	●	n/a	○
UbiREAL	●	○	n/a	●
SitCom	●	●	n/a	○
P-Vot	●	●	n/a	●
StarBed2	○	○	n/a	n/a

Table 2.3: Pervasive Computing Components Modelled

a generalised pervasive computing simulator should be able to simulate all sub-domains of the broad pervasive computing domain. To address this, an examination comparing the various approaches employed by the simulators is now presented. In addition to highlighting the strengths and weaknesses of each approach, alternative approaches that may not have been considered by the original seven simulators are introduced, supplementing those approaches already examined. Thus, it is hoped that a more complete set of approaches is identified, along with any difficulties that accompany those approaches.

2.6.1 RC1 - Flexibility in Modelling Pervasive Computing Components

Table 2.3 shows the degree to which the simulators modelled the components that are typical of pervasive computing environments. Apart from the observation that none of the simulators model all of the components, the most notable feature is that almost all of the simulators, all with the exception of Tatus, fail to provide any support for actuators. This is most likely a reflection of the fact that the proliferation of sensors in pervasive computing environments has occurred at a much faster rate than that of actuators, thus generating a greater requirement for the modelling of this functionality.

Environment

Three approaches of note have been introduced by the simulators discussed. Using a graphical tool, Ubiwise, Tatus, and P-VoT can specify a model of the environment in a realistic 3-D format. From a person's perspective, this seems ideal, as it is far more tangible than specifying

the environment in terms of a physical quantity, which is the approach adopted by UbiREAL. However, the lengthy time required to develop a model of a physical space makes this approach prohibitive, even though it is adopted by many simulators. Another point to note is that it has been previously time-consuming and difficult to expand these models into larger and more detailed representations of the environment, although tools are emerging that will assist in the automation of this task in the future.

Specifying an environment in terms of a physical formula is more expressive from the point of view of the phenomenon being measured. A physical formula is also more flexible when considering the interaction that components can have with that environment. For a sensor to calculate the value of the environment at a point in the modelled space, it simply has to plug in some values, such as its location, into the physical formula specified.

What is common amongst these two approaches is that they both support a publish-subscribe mechanism to notify sensors of changes in the measured phenomenon. Similarly, game engines typically provide a trigger mechanism, which can be exploited for this purpose.

The final approach is to use a simple set of key-value pairs. This approach has been used in SitCom and also in work by (Martin 06b) in the development of their simulator for context-aware computing. In this approach, some aspect of the environment, whether a physical phenomenon or a sense-able "thing", is indexed by a key and the value of the sensed data is returned. This approach is limited to phenomena that are simple in type, and do not vary across more than one parameter. For example, in order to simulate a phenomenon across a physical space, in the way that the physical quantity simulators such as UbiREAL do, a key-value pair would have to be generated at every location, which is inefficient.

(Naumov 03) have described an extension to ns-2 (Fall 01), that improves the modelling of the environment with a view to improving the efficiency of a simulation. Naumov et al, have proposed partitioning the space tracking the location of nodes into a grid comprising physical spaces. This logical partitioning enables a more efficient calculation of the location of nodes within the network simulation, and has been shown to improve the performance of the simulator by significant factors. A similar approach is used by (Sundresh 04) in implementing their model of an environment in their wireless sensor network simulator, SENS. The model

of the environment is partitioned into smaller physical spaces, whereby each space, called a tile, assumes a value of either grass, concrete or wall. The model of the environment is used to model more accurate radio-propagation models for wireless sensor networks.

Sensors

Tatus, UbiREAL and SitCom offer the most varied approaches to the modelling of a sensor. The most basic approach to modelling a sensor is to abstract the physical device to a query API. This approach is adopted by UbiWise and Tatus, whereby calls to a sensor API return sensor data from the graphical environment. The Lancaster simulator provides a slightly more complex API, that returns contextual data rather than raw physical data, but the query API abstraction remains the same. This simplistic model does not take into account any characteristics of the sensor itself.

A slightly more complex approach is to model the device driver that would be used to normally access a sensor, an approach that is used by UbiREAL. In this case, the sensor data returned is the same as that of UbiWise and Tatus, but it is returned in a format that the application is expecting. As is noted, using a simple query API means that applications have to be re-written to use that API. UbiREAL's approach of abstracting sensors to device drivers, essentially provides an emulated interface, on top of a basic query API. A similar approach is used in the wireless sensor network simulator domain. TOSSIM (Levis 03), and EmStar (Girod 04a) to a certain extent, provide the notion of an Analog to Digital Converter (ADC) channel, but only provide a simplistic implementation with port/value pairs, i.e. a port is read and a single 10 bit value is returned. This value can be random, or a general model can also be implemented using an external function.The resultant emulated sensor data is pushed to the emulated TinyOS application.

SitCom uses its 3-D game-engine based environment model to its advantage to generate accurate models of more complex sensor streams such as video feeds or audio streams. This is a great example of exploiting the graphical model of the physical environment to its full potential, something that UbiWise and Tatus arguably do not.

What is missing from all of these models is capturing the actual device itself. When an

API is presented to an application as an abstraction of the sensor, all of the characteristics of the sensor are missing. Different sensors have different thresholds for example, and behave differently according to an error model. Capturing this error and these sensor characteristics can be useful in the complete modelling of the domain. In wireless sensor networks, modelling what happens at the physical sensor is often neglected, because the focus of the research is often on the network layer or the query. There are, however, some wireless sensor network simulators that offer more complete models of sensor devices than the simulators in the pervasive domain, and is more reflective of the error-prone data that sensors provide in reality.

SensorSim (Park 00) introduced the notion of a "sensor channel", which models the phenomena which is sensed rather than the channel for delivery of sensor events. A sensor channel exists that models the wave propagation characteristics of infra red light, and a separate model exists for representing sound waves. In J-Sim (Sobeih 05), the notion of a sensor channel is again used to model senseable phenomenon. A sensors physical layer is used as an interface to this channel and is the sole input into the sensor channel. A *sensor propagation model* is used in the sensor channel. Signals propagated over the sensor channel may be attenuated, according to signal strength, receiving thresholds and other characteristics determined by the sensor propagation model. Other factors taken into account in the model are the location of the sensor, and that the location of other sensors may also impact upon the signal propagated over the channel.

Actuators

The only pervasive computing simulator to address the model of an actuator is Tatus, and similarly to its model of a sensor, it is abstracted to an API rather than modelling the device itself. The simple API offers applications the opportunity to update some state in the world, the effects of which are captured within the graphical environmental model. However, this technique offers no opportunity to specify actuators with different characteristics, that might affect the environment in different ways.

Actuators, however, are more common in the robotics community, and a simulator developed by (Labella 06) for the sensor and actuator network (SANETs) domain has merged

a robotics simulator and a wireless sensor network simulator. The ODE (Ordinary Differential Equations) system (Parker 04), models the rigid body dynamics of a physical robotic actuator. Similar research work has been achieved in Player/Stage[11] and has recently been applied (Kranz 06) to the pervasive computing domain. This library approach to modelling actuators is however constrained to robotics, and is not extensible to actuators, for example light switches, or radiators, that are often considered in the smart-space domain, a subset of pervasive computing. From a more general perspective, it would be useful to consider actuators as they might apply to a physical environment. In particular, UbiREAL's approach of specifying the environment in terms of mathematical functions defining the characteristics of the space would appear to overlap with the robotics approach. Given the expertise required to generate the formulae of the phenomenon, the formulae could be extended to accommodate updates to the model based on the actions of an actuator.

Users

Two approaches are evident in modelling the users within a pervasive computing environment. UbiWise, Tatus and UbiREAL all exploit the functionality of the underlying game engine to provide realistic models of human users that can interact with their respective modelled environments. Gamebots and Non-Playing-Characters (NPCs), can be programmed to interact with their environments, including moving around the environment, using objects, such as applications (in Tatus) or physical devices (in UbiWise) modelled within that environment, and interacting with other NPCs. The game engine API simplifies the task of modelling the functionality of users within the model.

It seems clear though, that the approach of using an NPC is dependent on the physical environment that is captured by the game-engine. In wireless sensor networks, the user is usually defined in a strict mobility patterns or more recently, a trace-based mobility patterns. In intelligent transportation systems, the user is a vehicle or a person inside a vehicle. Considering this broader range of pervasive computing domains, then individual NPCs must be created for each domain.

[11] http://playerstage.sourceforge.net/

P-VoT specifies the behaviour of users in terms of a Statechart, i.e. modelling states, and state transition functions. More complex behaviour can be specified in the Python scripting language, although there is no specific code base for modelling users.

There are two points to note. The first is that the model of a user is dependent on the model of the environment in which that user exists. Secondly, that model of the user must interact with that environment in a specified way. This means that the model of the physical environment, must be explorable or queryable from a user's perspective. This holds true whether the model of the environment is captured as a building, or a series of rooms, as is the case in UbiWise and Tatus, or if the model of the environment is captured as a series of states, in the case of P-VoT.

Several models from outside the pervasive computing space have been applied to modelling users. Formalised models such as random walk and random waypoint (Camp 02) have been long established and used within the wireless network simulation community. These are, however, basic models of a user's location, moving around an open space, whose pattern of movement is often not affected by their environment. More complex mobility patterns have been established for particular transport domains such as rail (Li 06), and road traffic (Esser 97) simulators, and these models capture an increased dependency on the environment compared to the classical mobility patterns.

In addition to these mobility patterns, efforts have also been made to capture the behaviour of users specifically in the pervasive computing domain. (Narendra 05) has specified a user's behaviour in terms of event-condition-actions, a rule-based system that has its roots in multi-agent simulation. The complete system models hardware devices as resources and applications, and they are also specified in terms of event-condition-actions. The environment itself is presented in terms of contextual information. An alternative approach to modelling the user using XML has been addressed by (Heckmann 03), who exploits an ontology language, UserML, to model attributes of the user such as their location, their actions, and the conditions they are acting under. This model is to be used by real applications to exchange data about users though, rather than being part of a larger simulation model, so unfortunately it does not tie into any specific models of the environment that a user might exist in.

2.6.2 RC4 - Application Emulation

A typical pervasive computing application requires emulation on two fronts. An application can use sensors and actuators to interact with its environment, and it can also use a network, either wired or wireless, to communicate with its peers. There are two main approaches to achieving the emulation of these two interfaces.

The first emulation technique, adopted by UbiREAL, and StarBED2 uses a compile-and-link approach. In this approach, the target application is emulated at the immediate level of its interaction with its underlying execution environment. A library is written that conforms to the API of the system calls or middleware, but interfaces with the simulator instead of the original target platform. The emulated application is compiled and linked against the simulators library instead of the original libraries.

A similar approach is used by LSE to enabling the emulation of the application's networking functionality. However, instead of redirecting method invocations within the application, the application is compiled and linked as normal, and the kernel is modified to intercept messages and re-direct them into the network simulator.

The decision of what level to emulate at is dependent on a variety of factors, and may be influenced by whether it is the network interface or hardware interface, or both, that is being emulated. Additional important factors include the availability of the source code for the application, middleware and system calls upon which the application is built, and the accuracy of emulation required. A compile and link approach to emulation is only possible when the source code of the application is freely available. In this case, intercepting calls within the kernel is possible, but may only be suitable for emulation of the network interface.

An additional factor to consider is the accuracy of the emulation required. In the compile-and-link approach, calls to the underlying functionality, whether it is system calls or middleware are redirected into the simulator. If an application is built upon middleware, and calls to that middleware are redirected immediately into the simulator, then all of the functionality of the middleware is lost. If the source code for the middleware is available, it may be more appropriate to retain some of the functionality, and redirect methods to the simulator after some of this functionality is invoked.

2.6.3 RC6 - Network Simulator Integration

There are two approaches to supporting network simulation within these simulators: build your own or integrate an external simulator. UbiREAL and Tatus have developed their own network simulators, which have been integrated directly into their simulators. There are positives and negatives to this approach. There are well established network simulators, such as J-Sim (Sobeih 05), which are already widely supported by the wireless sensor network community. Developing yet another network simulator is a time consuming task, and is an area of research in its own right. However, designing your own simulator does allow the developer to exploit factors specific to network simulation in pervasive computing environments. Tatus, for example, factors in the location of rooms and walls, provided by its world model, to model more complex wireless networking environments than a standard network simulator.

An alternative approach is to integrate an existing network simulator into the simulator to perform the role. This approach, used by LSE, requires that mappings from the pervasive computing simulator to the network simulator is provided. For example, the location information of the individual nodes may be modelled by the pervasive computing simulator, but is needed by the network simulator for the calculation of nodes within a transmission range for example.

An additional factor for consideration is that if the applications are being executed in wall-clock time, then there may be limits on the number of nodes that an external network simulator can model. The network simulator has two tasks. The first is to calculate the receivers of a message that is sent, and the second is the delivery of that message to the correct receiver at the correct time. If the time taken for the two of these tasks is greater than the time taken to deliver the message in the real world, then the simulation of the network is inaccurate. The time taken to calculate the receivers increases substantially as the number of nodes in the simulated wireless network increases, so this may be a limiting factor in this approach.

2.7 Summary

There have been numerous simulators that have been developed to address the need for evaluation tools for pervasive computing. UbiWise, the successor to QuakeSim, was the first of the well known simulators to be developed, and its game-engine based approach has been utilised by further simulators in recent years, notably Tatus and SitCom. More recently, UbiREAL has recognised the importance of a more comprehensive approach to simulating the domain, integrating improved models of the physical aspects of the environment.

However, these simulators have only met with limited success, and none have addressed all of the requirements identified for modelling a domain. Perhaps the most striking area of this is in the modelling of the components that are the very fabric of these pervasive computing scenarios. None of the simulators discussed attempt to model a physical sensor itself, instead they prefer to abstract it typically to a query interface or a device driver. There is less support for actuators in the existing state of the art. One of the exciting prospects in pervasive computing is the use of actuators to affect the environment, yet this interaction between the environment, and the actuator, and the actuator device itself is rarely modelled.

The lack of models for these devices is partly due to the fact that pervasive computing is a very broad area, that encompasses a wide array of physical devices. It is difficult to come up with a set of components that are representative of the domain, and that may interact with each other in unanticipated ways. There have been attempts to integrate third party simulators that address the modelling of these components individually, but without an infrastructure to bind these components together, this approach can only ever have limited success.

Although it has been established that pervasive computing is a broad multidisciplinary domain, the underlying components such as applications, sensors and the model of the network are in fact constrained at an abstract level. For example, it has been noted that several simulators, UbiREAL, Tatus, and UbiWise all employ a publish-subscribe design pattern or similar to notify sensors of changes in the environment, and that this design pattern is independent of the implementation of the sensor. Similarly, it has been noted that in pervasive computing the actions of the user, who may be carrying devices or applications, are influenced

by their environment. Compared with the wireless sensor network simulator domain, where a user is defined by a strict mobility pattern, more realistic models of the user and their interaction with their environment are required.

This chapter has provided an overview of the state of the art in the area of simulators for pervasive computing. Using the review criteria determined, the main works in this area have been examined, and have been shown to be deficient in several aspects. Finally, a comparison of the adopted approaches to modelling the main components was presented, and additional related work in these areas were also identified.

Chapter 3

The PiCSE Framework Architecture

3.1 Introduction

As presented in the opening chapter, this thesis describes a framework enabling both the simulation and emulation of pervasive computing scenarios. The PiCSE framework is comprised of three class categories, the *PiCSE_Core*, *Abstract_Interfaces*, and *PC_Abstractions* that combine to form the PiCSE framework. Developers use the *PiCSE_Core* classes to support instantiations of the *PC_Abstractions* class category to create simulations of tailored pervasive computing scenarios. Additionally, the *Abstract_Interfaces* class category can be used to extend the framework to meet modelling requirements not provided by the *PC_Abstraction* class category.

This chapter presents the PiCSE framework and examines how the creation of simulations is supported by the PiCSE framework and the underlying infrastructure. The chapter begins by justifying the applicability of the framework-driven approach and identifies requirements that will guide the framework's design. The conceptual architecture of a typical PiCSE instantiation is then considered. This highlights the main levels of interaction between the respective components of an instantiation. The main abstractions that are provided to model these components are then introduced, and this is followed by presentation of the framework's support for the realistic emulation of applications. Finally, the overall architecture of the *PiCSE_Core* class category is examined.

3.2 A Framework-Driven Approach for the Testing of Pervasive Computing Applications.

Based on the core challenges and the seminal definitions for frameworks introduced in chapter 1, three criteria should be satisfied in order to consider a framework driven approach for the testing of pervasive computing applications. These are:

1. Are these a family of related problems?
2. Do they interact in a defined manner or pattern ?
3. Could the current approaches benefit from avoiding re-creating and re-validating common solutions to recurring application requirements and software design challenges?

The answer in all three cases is yes.

Are these a family of related problems? Currently, there are a wealth of simulators and emulators all addressing different but related aspects of the pervasive computing domain. Within these simulators and emulators, there are many recurring actors, components and elements including persons, sensors, locations, communication abstractions and the environment and in many cases these aspects are conceptually similar but are simulated to different degrees of granularity or complexity depending on the aim of the simulator. For example, in a network simulator such as NS-2, a person is abstracted to a mobility pattern such as random walk, or random waypoint (Camp 02), whereas in a context-aware computing simulator, a person may have additional features such as an identity, a current activity, and a plan associated with that mobility pattern. In this case, clearly the concept of location, mobility and a person are related, but these are implemented using different models and complexities across the different simulators.

Do they interact in a defined manner or pattern ? The interactions between elements that are recurring between simulators are not necessarily pre-defined across all scenarios but there are recurring interaction patterns between them. For example, sensors often take sensor

readings from their local environment, and those sensor reading are generally used by some application for the purpose of influencing and driving some application behaviour. Environmental monitoring applications often use static wireless sensors and a gateway application may distribute a query to those sensors to gather data within the sensor network. Alternatively, a sensor can be event-triggered by a change in the environment, for example a person entering a room or a proximity sensor that detects when a door has closed, and these events can trigger some reactive application behaviour. A second alternative interaction is that an environmental monitoring sensor can be hosted on a mobile device which is carried by a person, and is only invoked when the person is using a context-sharing application. In all three scenarios, wireless sensor networking, smart-spaces, and citizen sensing, there is a recurring relationship, a pattern, between the sensor, the environment and the application, although the actual mechanics and nature of the pattern differ in each scenario.

Could the current approaches benefit from avoiding re-creating and re-validating common solutions to recurring application requirements and software design challenges? At present, there are a wide range of simulators for each of the pervasive computing sub-domains, and each has their own interpretation and implementation of the recurring elements such as location, sensors, events, and mobility models. Avoiding the re-implementation of these components would both ease the task of development, and would give greater credibility to any simulated results as those results would be built upon a common base. Finally, a set of components and abstractions that were recurring across all pervasive computing domains would open up new opportunities for evaluation of overlapping domains as well as providing a base for an extensible architecture that could account for emerging domains in the future.

3.3 A Frame of Reference for a Pervasive Computing Simulator

In chapter 1, a broad overview of pervasive computing and its modern definition was presented. In recognising that the future of pervasive computing will be both multi-disciplinary and dynamically changing, the focus of this thesis's approach is on an extensible architecture that can accommodate the modelling and evaluation of both new and existing sub-domains of

pervasive computing. As such, the PiCSE architecture is not targeting a sub-set of existing pervasive computing domains and instead intends to be at least theoretically applicable to all pervasive computing domains.

The thesis's approach considers the user to be a researcher or domain specialist who is using a standard commodity personal computer, i.e. without access to high-performance computing hardware. There are no unusual restrictions on the platform that the framework should be deployed on, but the "typical" use-case that we are considering is that of a researcher performing some rapid prototyping, iteration and evaluation of a pervasive computing scenario and as such is someone working at his own machine, typically a desktop computer.

3.3.1 The Scope of External-Facing Features

The analysis of the most pertinent state of the art simulators in this domain, outlined in chapter 2, revealed a range of recurring features that are common to pervasive computing simulators. These included the modelling of sensors, users, domain-specific applications, the environment, and network simulation amongst others.

Network Simulation

The design and implementation of the framework focuses around the core challenges of flexible heterogeneity, extensibility and reusability. One recurring component of pervasive computing simulators is the aspect of network simulation. There are already many well-established network simulators in this domain, and the PiCSE approach does not attempt to create a new network simulation component that could compete and potentially displace one of these. Rather, it is recognised that the research and development of such network simulators is an entire and independent field of research in its own right, and PiCSE's framework implementation does not attempt to duplicate this research. Instead it focuses on an extensible open architecture that allows the future integration of a open network simulator, and thus combining the best of both approaches.

User Interfaces

The simulators outlined in chapter 2 implemented a variety of user interfaces ranging from a shell-based command-line interface up to a 3-D interactive graphical user interface used for both the definition and the execution of a simulation. The overall system is implemented as a series of libraries and associated scripts for the building and executing of simulations. Not having a graphical component means that experiments that don't require a human in the loop can be executed quickly. Including a "human in the loop", i.e. in the execution of a simulated experiment is an inherently slow approach that is not suited to large scale experiments. However, PiCSE's flexible and open implementation ensures that the architecture could be extended in that direction through the addition of a graphical user interface if that was required.

The Human or User Factor

Nevertheless, the human or user element and behaviour is a key part of many pervasive computing scenarios, particularly those involving smart spaces or context-aware computing. The modelling of user behaviour within PiCSE is defined using a combination of both mobility patterns and event-based model. Although it does not define user behaviour in a manner as extensive as an agent-based model, the basic user model can interact, sense, move within and affect both the modelled environment and other elements with that modelled environment such as sensors or other domain-specific elements. This approach allows PiCSE to handle users as it would any other modelled element within a simulation. In line with Weiser's original vision of pervasive computing, PiCSE's support for application and device emulation is restricted to those devices and applications that run in the background, sensing, reacting and ultimately prompting changes in their environment. Therefore, the architecture described hereafter only considers applications that can run as discrete services and not those that that may require direct user input.

As will be seen however, many of these restrictions that apply to the scope of the framework's architecture do not preclude the integration of these components and features at a later date. Where appropriate, these extensions are discussed inline in the text.

3.4 Requirements

There were three main factors taken into consideration when defining the requirements for a pervasive computing simulator. These were primarily highlighted by the analysis of the state of the art. Review criteria such as *flexible heterogeneity* and *extensibility* are only partially met by the current state of the art and are thus included directly as requirements. The requirement of a simulators support for *external interaction* was omitted, as the focus of the thesis was to develop a standalone extensible architecture that could run on a local machine. Secondly, the evolving multi-disciplinary and emerging domains of the future of pervasive computing motivate and provide further requirements. Finally, the frame of reference for a pervasive computing simulator, as defined in the previous section also influences the identification of the requirements.

There are many simulators that address subsets of the complete pervasive computing domain but few are generic enough to address all of these subsets. Often, these simulators are not suitable for evaluating application code but fortunately there are platform-specific emulators that meet this requirement. However, these emulators often over simplify the inputs and outputs of the emulated applications, and it is a requirement of this framework that the flexible and extensible modelling of these sensor inputs and actuator outputs are a fundamental part of the complete evaluation of pervasive computing scenarios. Section 3.2 has stated that by identifying recurring commonalities within the domain, using a framework is a suitable approach to supporting the wide range of related pervasive computing scenarios. The following requirements are therefore identified for the PiCSE simulator:

- **R1** The framework should support the flexible and heterogeneous simulation of the hardware elements of pervasive computing scenarios. This is be provided by creating customisable abstractions representing physical aspects of the scenarios such as the environment, as well as hardware devices such as sensors and actuators.

- **R2** The framework should support the emulation of real-code applications developed for pervasive computing. These applications may be built upon middleware that should also be supported.

- **R3** The framework should mediate the flexible interaction of any simulated and emulated elements forming the simulation of a pervasive computing application.

- **R4** The framework should support the creation of new abstractions of both hardware and software elements, in order that future emerging application scenarios can be incorporated into the PiCSE framework.

- **R5** The framework should combine recurring elements of pervasive computing simulations into a set of reusable components thus reducing the amount of work required to instantiate any simulated scenarios.

- **R6*** The framework should support network simulation for both wired and wireless networks.

- **R7** A framework that supports the creation of pervasive computing simulations must provide functionality to support the evaluation of those simulations.

At present, there are no simulators for the pervasive computing domain that meet all of these requirements. A simulator designed as a framework could potentially meet these requirements and would form a contribution to the state of the art in this area.

Requirements R1, R2, R3

Requirements R1, R2, and R3 all address the core challenge of flexible heterogeneity. A set of modelled hardware and software elements that were recurring across all pervasive computing domains and that could be freely combined would open up new opportunities for the evaluation of many pervasive computing domains including future domains where the boundaries of pervasive computing and other domains begin to intersect. At present this level of flexibility is not provided by any of the state of the art simulators and this requirement is mandated by the future direction that pervasive computing is moving towards.

Requirement R4

Requirement R4 maps directly to the core challenge of supporting extensibility. A framework should support the ability to accommodate new emerging application areas of pervasive computing, and accommodating these areas should be reduced by leveraging on the previous work encapsulated within the framework, i.e., the task of accommodating these new areas should be simplified. Without this requirement, new modelling tools must be developed in conjunction with each emerging application area, despite the predicted recurrence of many of the aspects of the emerging domain with aspects of previous pervasive computing domains.

Requirement R5

The requirement R5 maps directly to the core challenge of reusability. Given the recurring elements, behaviours and patterns that have been identified in pervasive computing application scenarios, these should be encapsulated within a set of core components within the framework that are common to all instantiations of the framework. This requirement follows on from requirement R4 and is also mandated by the framework-driven approach that has been deemed to be the most applicable to this domain.

Requirements R6* and R7

Finally, R6 and R7 are two functional requirements for the framework. These requirements were identified as necessary by their frequent recurrence in the simulators examined in the state of the art, and address both key functionalities and usability. As noted however in the previous section, although network simulation is a recurring feature of pervasive computing simulators, the approach used is to allow the integration of an existing and proven network simulator as a sub-component or an out-sourced feature of the architecture.

3.5 The PiCSE_Core Architecture

The *PiCSE_Core* class category provides the core and recurring components of an instantiation of the PiCSE framework, i.e., a simulation. It controls the dispatching of events raised

by the simulated models and controls the execution of applications in the form of pthreads. Figure 3.1 identifies the key components within the *PiCSE_Core*. The *PiCSE_Core* itself is implemented as a collection of concrete static classes, and can be logically divided into two sections: the majority of the components interact with both the *Simulated_Models* and *Emulated_Interfaces* layer through the PiCSE API, whilst the `Domain_Manager` (including its sub-component, the location manager) interacts solely with the *Simulated_Models* layer.

Each of these components plays an important role in the instantiation of a PiCSE instantiation. The `Domain_Manager` is responsible for the management of instantiations that represent the "physical" aspects of the domain. It provides the layer stack, the collection of independent `EntityLayer` and `EnvironmentLayer` instantiations, and maintains the dependencies between them. It provides a global definition of variables, such as the dimensions of the modelled domain, and provides services to the simulated components, such as generating random locations, for the instantiation of simulated `Objects`.

In addition to this, it also plays the role of a `Location_Manager`. The `Location_Manager` maintains the location of `Net_Interface` instantiations, which are used to provide a basic network simulation interface. As an `ExecutionEnvironment` instantiation moves throughout the modelled domain, the location of its respective `Net_Interface` also moves. The `Location_Manager` manages the modelling of their locations, which is required by the network simulation component.

The network simulation models the wired and wireless network communication of a PiCSE instantiation. The model supported is very basic: A packet (in the form of a `string` or a `void*`) can be broadcast to all local neighbours or can be unicast to a single neighbour. Packets can be "dropped"probabilistically, and also based on transmission range. Both of these parameters can be set by the user. The calculation of the distance between potential receivers and the sender is performed in conjunction with the location manager, which maintains the location of all `Net_Interface` objects. The realistic modelling of a network simulator is not a feature of the PiCSE simulator. There are many existing network simulators, and the development of another is an area of research in its own right. Instead, the PiCSE core has been architected bearing in mind the future potential integration of a well-known network

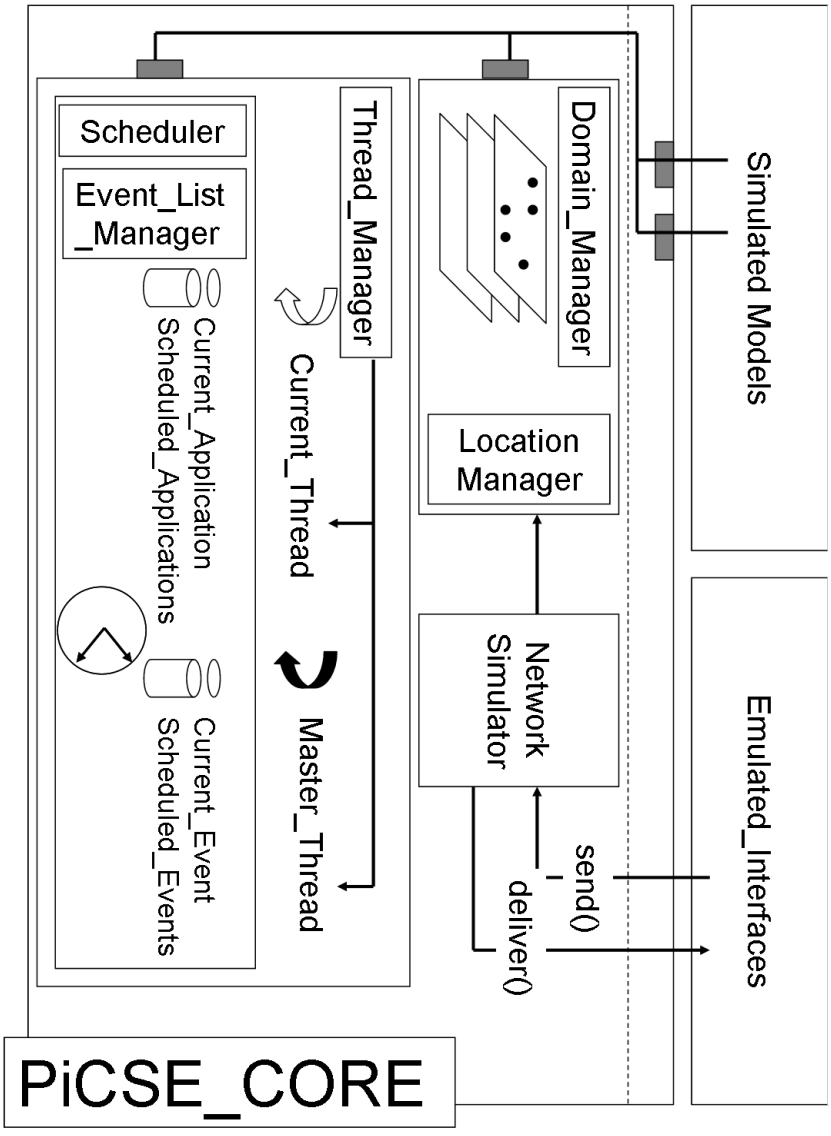

Figure 3.1: The PiCSE_Core architecture

simulators, such as GlomoSim[1] or Opnet. Although this has not been evaluated in any of the evaluation scenarios, the separation of the `Location_Manager` and the `Net_Interface` abstraction from the network simulation component ensures that the network simulation component itself could be potentially replaced.

The `Thread_Manager` and the `Event_List_Manager` perform the joint role of executing a PiCSE simulation instantiation between them. As mentioned previously, the framework's support for emulating multiple applications is implemented using a master-slave multi-threaded approach. The `Thread_Manager` manages the controlled execution of both master and slave threads. The master thread itself is used not only to switch between the slave threads, but also to advance the `Event_List_Manager`, which is responsible for the scheduling of events originating from both the simulated models and emulated applications. Slave threads are scheduled to wake up at a time determined in advance of the thread being put to sleep.

The `Event_List_Manager` maintains two lists: the `Scheduled_Event` list and the `Scheduled_Application` list. The `Scheduled_Event` list is an implementation of a classical DEVS event list: a set of events ordered by their timestamps. The `Scheduled_Application` list is a set of `<pthread, ExecutionEnvironment*>` tuples, also ordered by their timestamps. The master thread advances the two event lists, by checking and determining the next event. In the case of the next event coming from the `Scheduled_Event` list, the thread of control remains with the master thread, the event is dispatched, and the appropriate functionality of the simulated entity is invoked. Alternatively, an event is dispatched from the `Scheduled_Application` list. In this case, the `pthread` contained within the tuple is restarted as a slave thread, and when the application has completed its execution cycle, the thread of control is returned to the master thread. In the case of either event type occurring, the current event is stored as a global variable for the duration of the execution of the appropriate methodology.

Emulation transparency is maintained as both the behaviour of interacting hardware devices and the timing of any behaviour is captured within this model . Further detail of the behavioural interaction and transparency is outlined in sections 4.3.5 and 4.4.1 of chapter 4.

[1] pcl.cs.ucla.edu/projects/glomosim, www.opnet.com

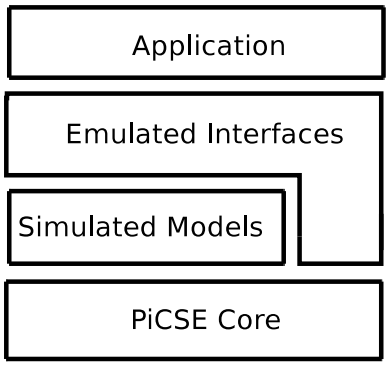

Figure 3.2: Conceptual architecture showing layers within a generic instantiation of the PiCSE framework

3.6 The Composition of a PiCSE Instantiation

The framework supports three levels of modelling: the creation of simulated models only, emulated applications only and both together. Figure 3.2 represents a conceptual architecture capturing the most generic instantiation of the framework involving both simulated and emulated entities. The purpose of this is to have a frame of reference when discussing the interaction of components between these layers. The following conceptual layers are identified and now described from the bottom up.

The PiCSE Core Layer

The *PiCSE_Core* provides the main functionality required to support and manage all instantiations of the PiCSE framework. This includes, amongst others, an event list, a scheduling API, an event manager, an object manager, a thread manager, and a network simulation component. The functionality here is provided by components within the PiCSE_Core class category, and are discussed in detail in section 3.5.

The Simulated Models Layer

The layer directly above the *PiCSE_Core* is the *Simulated_Models* layer. Classes within this layer are either derived from the abstractions provided within the *PC_Abstractions* class category or are domain-specific models derived from the *Abstract_Interfaces* class category.

Simulated models from a typical pervasive computing scenario would include sensors, actuators, a model of the environment and domain-specific models. These interact with the *PiCSE_Core* to schedule events, thus driving the simulation. They can also interact indirectly with other simulated objects through the specification of both physical and logical dependencies, which are described in detail in chapter 4.

The Emulated Interfaces Layer

The *Emulated_Interfaces* layer is the next conceptual layer and is required when applications form part of the modelled scenario. This layer sits between the simulated models and applications and mediates the interaction between the two. Classes within the *Emulated_Interfaces* layer are written to replicate the behaviour of the native environment in which the application would run and provide the mapping between a real application and simulated models. This can occur in two situations:

1. The first situation is when an application has been written to interact with a hardware device such as a sensor. In the case of a sensor, for example, an application may interact with the device physically via a serial cable, and logically, through the underlying file system of the operating system. In this situation, the simulated sensor must provide an emulated interface so that the application can read data from it in a particular format. Modelling of these interfaces is performed through instantiation of classes from the *PC_Abstraction* class category.

2. The second occurrence is when applications are built directly upon system calls or middlewares, and is shown in figure 3.2 as the direct meeting point between the emulated interfaces layer and the *PiCSE_Core* layer. In the case of a middleware providing networking functionality, the emulated interfaces layer provides a mapping between the application's calls to the middleware and the framework's network simulator. A library containing mappings of common system calls is provided in the *PiCSE_Core* class category, and users can instantiate their own application specific mappings using abstractions provided in the *PC_Abstraction* class category.

As noted above, it is possible to create instantiations of the framework without applications. It is even possible to create instantiations of the framework that include applications and are built without emulated interfaces. But in these cases, the application must be at least partially re-written to use the PiCSE APIs to access the simulated models.

The Application Layer

The final layer is made up of applications that are used within the scenario. The applications can transparently interact with hardware devices, middleware or the emulated operating system through the *Emulated_Interfaces* layer. Applications are used as they would be in the native environment, i.e. without change and are compiled into a PiCSE instantiation. Certain limitations exist that restrict the class of applications that can be run within PiCSE, and these limitations are explored later.

3.6.1 Minimum Requirements for a PiCSE Instantiation

There exists a set of minimum requirements for an instantiation of the framework. The framework supports the dual functionality of simulation and application emulation, each of which addresses the modelling of separate, yet inter-dependent, aspects of pervasive computing scenarios. It may be more helpful though to imagine the framework covering these two functionalities and the wide span between them. A typical pervasive computing scenario would contain elements of both simulated and emulated components. However, at the boundaries of this supported span are instantiations of the framework where only simulated entities or emulated applications are required. The framework enforces no restrictions or dependencies on the number of instantiations of simulated and emulated models. Regardless of whether the scenario being modelled requires simulated or emulated elements, or both, an instantiation of the *PiCSE_Core* is required. The provision of the *PiCSE_Core* that is reused in all variations of PiCSE instantiations partially address R5, the reusability requirement.

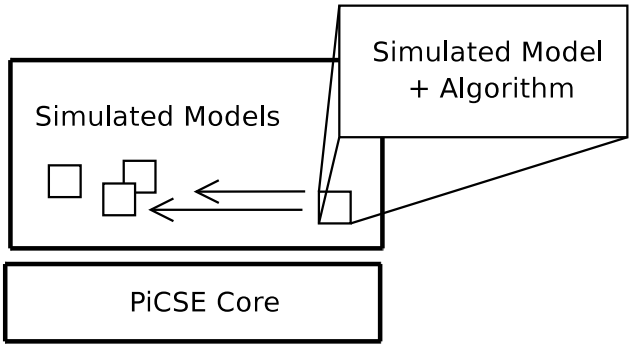

Figure 3.3: Conceptual diagram of a PiCSE simulation in Algorithm Mode

Algorithm Mode

In instantiations with only simulated models, models are created of the hardware devices, the environment and additional domain-specific objects (or a subset of these three). As there are no applications to interact with these devices, the objects can only interact amongst themselves. An API exists, within the class definitions of the key abstractions, that allows other simulated models to query the sensors as simulated objects, although not through an emulated interface that an application would typically use.

Within the context of the use of the PiCSE framework, this is depicted in figure 3.3. This is so called as typically a new object is introduced that implements an algorithm, that interacts with the hardware devices and the environment. The inclusion of an algorithm object is not necessary, but a simulation of hardware devices and the environment on their own, without any interaction, is not very interesting! New classes that implement algorithms are created using interfaces and abstract classes from the *Abstract_Interfaces* class category and not from the *PC_Abstraction* library.

Application Mode

At the opposite end of the supported spectrum are instantiations of the framework in Application Mode[2]. In Application Mode, shown in figure 3.4 , applications use an emulated interface

[2]No changes have to be made or parametrised within the framework to achieve the different modes of use. The concept of different modes is purely conceptual and merely reflects instantiations created with or without certain abstractions.

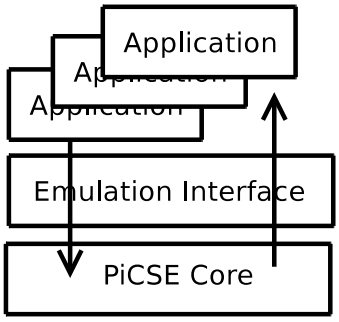

Figure 3.4: A PiCSE instantiation in Application Mode

to interact with the PiCSE core. In this mode, there are no simulated models to interact with, however the networking simulation functionality of the *PiCSE_Core* is available, allowing applications to exchange messages through the simulator. The Application Mode is highlighted merely to show that the framework supports the emulation of applications independently if required.

3.6.2 PiCSE Instantiations as Sub-Frameworks

The framework plays two different roles: The primary role, as outlined in Chapter 1 is to create simulations through the instantiation of the framework, whilst the secondary role of the framework is that it can be used to instantiate domain-specific simulators which are frameworks in their own right. These can then be used to instantiate simulations for a particular sub-domain of pervasive computing.

The many domains of pervasive computing, such as ambient computing, context-aware computing, and smart spaces, each have their own modelling requirements, such as the scale and type of environment to be modelled and the various hardware components found within such environments. Take for example, an intelligent transportation system (ITS) scenario. The required simulated models for this scenario might include the following:

- GPS and inductive loop sensors
- Road network environment
- Traffic light actuators

- Vehicles

It is entirely feasible to create an instantiation of the framework to model this scenario. This is in itself the primary role of the PiCSE framework. However, a single instantiation of the framework is restricted by the functionality of that instantiation. For example, the vehicle behaviour may be fixed or the behaviour of the traffic lights could be fixed. Different developers, that wish to evaluate separate applications within this domain, should be accommodated. One application might be examining the use of GPS sensor data in maximising vehicle throughput. A second application might look at using the same sensor data to provide collision avoidance software within the vehicles. In these cases, it makes sense to allow users to instantiate the PiCSE framework to create an intermediary framework that may be then instantiated to the individuals requirements. The result of a PiCSE instantiation in this case is not a simulation, but a framework allowing other developers to create their own ITS simulations. The PiCSE framework is flexible enough to allow multiple applications with different requirements to run within one simulation, should such a scenario be required by a developer. This is a significant step towards achieving the requirement of R1 and R2, the requirements relating enabling the flexible simulation and emulation of multiple elements of heterogeneous pervasive computing domains.

From an implementation point of view, the *PiCSE_Core* components remain the same. A new class category of abstractions and concrete classes, that are specific to the ITS domain, are created using instantiations and derivations from the *PC_Abstractions* and the *Abstract_Interfaces* class categories. Abstractions of this new class category are then instantiated to create individual concrete simulations. The ability to create new frameworks from the *PC_Abstractions* and the *Abstract_Interfaces* class categories contributes to meeting requirement R4, the extensibility requirement.

3.7 Supported Abstractions

The "generalised" PiCSE instantiation that is presented in section 3.6 has provided an introduction to the general categories of components that PiCSE supports. These are formalised as

abstractions within the PiCSE framework, which can be partitioned into three separate class categories. The *PC_Abstractions* and the *Abstract_Interfaces* class categories meet PiCSE's modelling requirements and are now introduced in the following sections 3.7.1 and 3.7.2. The first section explores the abstractions required to support the modelling of the "physical" aspects of pervasive computing, such as an environment or a sensor. The second subsection introduces models that abstract the "software" components, the applications and middleware that might exist within the modelled domain. The abstractions presented in these sections form the basis upon which the *PC_Abstractions* are built which satisfies requirements R1, R2, and R3, the requirements for flexible heterogeneity.

The implementation and interaction of these components is explored in greater detail in chapter 4 and validated in chapter 5, whilst their integration in the overall PiCSE architecture has been introduced in section 3.5.

3.7.1 Supporting the Modelling of the Physical Pervasive Computing Components

The PiCSE framework is based on the discrete event simulation formalism. In this formalism, a simulated event is defined as something that occurs at a particular discrete time within a simulation. It is the occurrence of these events, the corresponding state changes, and subsequent changes to the event list that make up an entire simulation instance.

All physical abstractions within the PiCSE framework are derived from the `Simulated` abstraction. This means that they can all produce, schedule and process events during the execution of a PiCSE instantiation. An instantiation of a "physical" abstraction can schedule an event to occur at a specified interval in the simulated future. When this time is reached, the event instance is dispatched, and the instantiation, to which the event belongs acknowledges the event and performs some action. The `Simulated` abstraction enables this event processing to occur at specific instantiations of "physical" components, and these instantiations are required to implement a simple event processing function `::doEvent( Event )` that is invoked when the scheduled event occurs.

In PiCSE, an `Event` is defined locally, i.e. events are specific to a particular type of entity,

and the same event for a different entity may have a different meaning, and may result in different behaviour. In practise, they are represented simply as enumerated types, and a single enumerated list is usually specified per instantiation of a physical abstraction.

The Object class

Several key abstractions have been identified that are present to various degrees in many pervasive computing domains. These include sensors, actuators, and other domain-specific objects, such as a vehicle in the case of an ITS scenario, or a mobile node in the WSN domain. The most common attributes of these have been aggregated into the `Object` abstraction.

Location

All instances of the `Object` abstraction have a location, which can be either explicit or implicit. An `Object` instantiation with an explicit location is an `Object` that is independent of other `Object` instances. That is, it either exists by itself or it is in control of its own location. Alternatively, an `Object` can have an implicit location. In this scenario, the object's real location is the location of another object with which it is physically associated.

This can occur through the feature of object collocation: a feature that supports the creation of complex objects, through the aggregation and association of instances of specialisations of certain abstractions. In the case of the `Object` abstraction, it is possible to associate physical objects with each other using the `::addObject(Object* b)` method of the `Object` abstraction. The result is that the "added" `Object` instance (b from above) assumes the location of the `Object` to which it was added. This frequently occurs in scenarios, in which an `Object`, such as a sensor, is carried by a user. The sensor's location is implicitly that of the person holding it, for example, a vehicle carrying a GPS device. Figure 3.5 is a UML sequence diagram that outlines the interaction between a "robot object" and a collocated "sensor object". As the robot moves, each collocated object is notified of the movement and updated its own position internally, and notifies any other references to that location. EntityLayer instances, which are mentioned in the diagram are a method for referring and accessing physical objects and are discussed in detail in chapter 4.

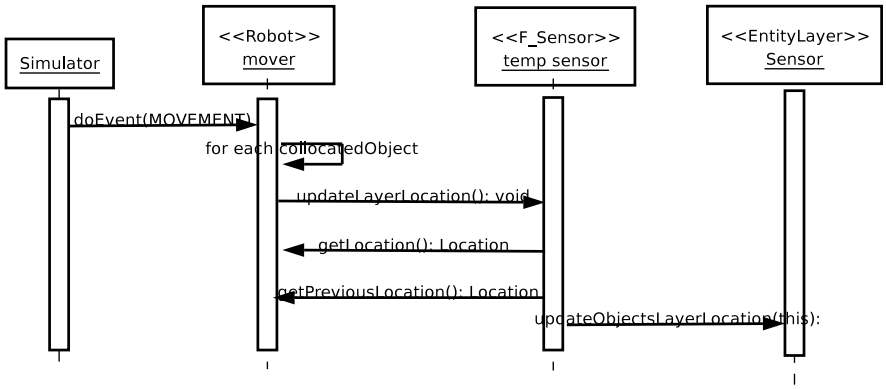

Figure 3.5: UML Sequence diagram depicting collocated objects scenario

Several related abstractions to the Object class are also provided. A `Mobility_Object` is an abstraction that provides mobility patterns, such as the random walk (Camp 02) and random way-point patterns, that are common in the simulation of wireless sensor networks. By implication, any `Object` instances that are "added" to a `Mobility_Object` instance would also move according to that mobility pattern.

`Sensor` and `Actuator` abstractions are also provided. Within the PiCSE framework, the `Sensor` abstraction is provided to model a physical sensing device, i.e. a device that can measure a physical phenomenon modelled within a PiCSE instantiation. As an object, it has a location, and that location can be used to sense an associated physical phenomenon at a certain point. A `Sensor` instantiation can act periodically or sporadically, and may be queried using the `::getState()` method. An invocation of this method implicitly uses the sensor's location to query the associated sensed physical phenomenon at that location.

Similarly, an `Actuator` can "affect" the state of a physical phenomenon at a certain location. It has a location, which again may be explicit or implicit, and can also generate events. A user of the PiCSE framework is required to define the effect that an Actuator can have on the associated physical phenomenon by implementing a `::setState( void* )` method. Location-based actuation on physical environmental phenomenon is supported, as well as actuation upon other `Object` instances.

Physical Domains

All instances of the abstractions introduced in section 3.7.1 are components of an overall model that captures a particular pervasive computing domain. The modelling of the physical aspects of the domain includes the capturing of many physical attributes such as measurable and manipulable environmental phenomenon, additional domain-specific features that may impact upon a model, and the objects themselves.

For example, a typical model of the domain of a smart space scenario may capture a user, any sensor that a user or application may utilise, a representation of a room or a building, and any physical phenomenon that the sensor may be measuring. If a user is to evaluate the performance or behaviour of the application, then a realistic model of the applications inputs (i.e., the physical phenomenon measured by a sensor) and outputs is required. Furthermore it is not possible to capture a building, or a road in the case of ITS scenarios, in an abstraction as broad as an `Object`.

To model the physical aspects of a domain, a new abstraction, `Layer`, is introduced. The `Layer` abstraction, and the related abstractions `Environment_Layer` and `Entity_Layer`, can be specialised to create models that support the flexible modelling of physical aspects of pervasive computing domains that cannot necessarily be captured by the abstractions already introduced above. The `Environment_Layer` abstraction can be used to capture environmental and spatial physical phenomenon whilst `Entity_Layer` abstractions can be used to capture objects that exist within those environments. These abstractions are explained in further detail in chapter 4.

Modelling physical phenomena using an extensible and flexible design is non-trivial given the wide range of phenomenon and environments encountered in pervasive computing. The `Layer` abstraction itself captures only the most general aspects of a domain, leaving the user with the task of building complex models through specialisation. In general however, a single instance of a `Layer` abstraction is used to capture a single physical aspect of a modelled domain. Conceptually, a layer is a 3-D grid of fixed size that maps onto the scenario being modelled. Layer instantiations are 2-D by default, but can support the third dimension of physical height. In addition to the physical x and y dimensions of the space, each layer

instantiation is parametrised with the granularity of the grid within the layer. The granularity of a "space" within the grid can be arbitrarily large or small and offers an additional degree of flexibility in the instantiation of the abstractions. The APIs that are provided to interface to the Layer exploit the discretisation transparently.

Even greater flexibility and complexity are achieved through the use of the layer stack. This is a representation whereby multiple layers representing individual physical aspects of the simulated domain are logically collocated and threaded together to build models that may be dependent on more than one phenomenon. This collection of layers is the entire representation of the domain being modelled, even though individual layers only capture a single phenomenon.

The interaction of both the `Sensor` and `Actuator` abstractions, with their associated physical phenomenon, is captured within the API of the `Entity_Layer` and `Environment_Layer` abstractions. In addition, instances of the `Entity_Layer` abstraction are used internally to optimise the performance of PiCSE, through the management of existing `Object` instances that a user may define.

3.7.2 Supporting the Modelling of the Soft Pervasive Computing Abstractions

PiCSE provides three main abstractions for supporting the integration of "soft" pervasive computing aspects. The `ApplicationWrapper` abstraction is used to integrate code from applications into a PiCSE instantiation. The `ExecutionEnvironment` abstraction is provided to emulate the environment in which the application originally would have been executed. Finally, the `Net_Interface` abstraction can be instantiated to access PiCSE's basic network simulator.

The `ApplicationWrapper` abstraction provides a wrapper that can be used to encapsulate an instance of an application. With some restrictions, it is possible to integrate application source code into a PiCSE instantiation without modification, allowing the source code to integrate with the simulated models introduced in section 3.7.1. This is achieved through the instantiation and association of an `ExecutionEnvironment` abstraction.

The `ExecutionEnvironment` may be viewed as either an emulated operating system or an emulated middleware, depending on the user's implementation. It supports multiple applications in the form of `ApplicationWrapper` instantiations, and also multiple simulated hardware devices, such as `Sensor` and `Actuator` instances, that may be associated with the application.

Communication is an important aspect of many pervasive computing scenarios, and many applications and middleware systems implement some sort of communication paradigm. PiCSE provides the `Net_Interface` abstraction to address this. The abstraction can be instantiated to implement both wired and wireless communication. Applications can use a `Net_Interface` abstraction, via an instantiation of their associated `ExecutionEnvironment` to broadcast or unicast messages across either a simulated wired or wireless network, and multiple network interfaces are supported.

3.8 Emulation support within PiCSE

Application code that can interact with sensors and actuators form an important part of the pervasive computing domain. It is a common, but an inefficient practise, that this code is typically written once for the simulation of a particular domain and is then rewritten at the time of actual deployment. This occurs because most simulators do not provide an API for the application being developed. PiCSE addresses this issue by providing the `ApplicationWrapper` and `ExecutionEnvironment` abstractions introduced previously. This section addresses how PiCSE supports this important functionality and thus meets some of the framework's most important requirements. Two aspects of the overall architecture that are worth highlighting are how specifically the framework supports emulation at the level of an individual application, and secondly, how the framework supports the emulation of many application instances.

3.8.1 Enabling Emulation

The following characteristics of applications deployed in pervasive computing environments are noted. Typically, they

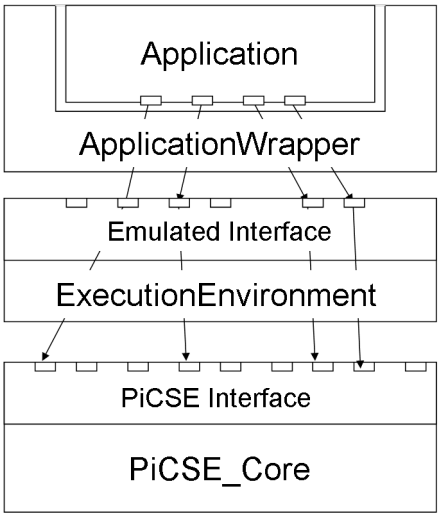

Figure 3.6: A split level implementation of the PiCSE support for emulating applications

- May interact with hardware devices such as sensors and actuators.
- May communicate with peers, servers and other devices, using a wired or wireless network.
- May be built upon middleware, that provides the above functions, or may access these services directly using low level system calls.

In the case of PiCSE, the hardware devices are implemented as instantiations of the abstractions introduced in section 3.7.1, and an API is provided to model the wired and wireless communication. Where effective emulation is required, it is imperative to maintain the APIs upon which the application is developed, whilst seamlessly interacting with the simulated models. The application believes (and behaves as if) it is interacting with real hardware devices, when it is actually interacting with instantiations of the PiCSE abstractions.

PiCSE achieves this using a split level design illustrated in figure 3.6. The application is integrated into the framework using the `ApplicationWrapper` abstraction. The API of the *PiCSE_Core* exists at the base level and provides an interface to all instances of the "physical" abstractions, such as the sensors and actuators, as well as the remainder of the PiCSE

functionality. An instantiation of an `ExecutionEnvironment` abstraction exists between the PiCSE core and application levels, and mediates the interaction between the application and the core. This `ExecutionEnvironment` instantiation must "bind" the calls from the application's original API to the corresponding functionality within the *PiCSE_Core*. The libraries, middleware or system calls, upon which the application is based, have to be rewritten in order to achieve this successfully. An instantiation of the `ExecutionEnvironment` abstraction plays this role. Using this methodology, PiCSE is in fact transparent to the application.

Although this approach requires some overhead in implementing the `ExecutionEnvironment` instantiation, it is envisaged that these "bindings" will only have to be written once per application type and that over time a library of common re-usable bindings can be produced, i.e. one binding for the TinyOS API, one binding to implement Linux system calls and so on. When a binding already exists for a common platform, it will be possible to integrate applications directly into PiCSE instantiations using only an `ApplicationWrapper` instantiation, which requires significantly less implementation effort. The split-level design partially satisfies requirement R4, software extensibility as it allows new software platforms to be integrated into the PiCSE framework in the future.

3.8.2 The Physical Architecture for Emulated Applications

An entire instantiation of the PiCSE framework consists of instantiations of the abstractions introduced in section 3.7, instantiations of abstractions derived from the *Abstract_Interfaces* class category all supported by a single instantiation of the *PiCSE_Core*. All of the aforementioned class categories and abstractions are implemented as libraries of C++ classes. Emulated applications in the form of source code, which must meet certain requirements that are now outlined, are compiled and linked into the libraries.

A point to note is that the process of application emulation begins with the command to compile the executable. As the application code is unmodified, calls within the application to certain methods must be intercepted. If they are not, then the code will compile and link against the applications original libraries. By using certain flags, the compiler is redirected to the emulated versions of the application's libraries. The linker completes the process

of binding the application calls to the emulated functionality of an ExecutionEnvironment instantiation.

PiCSE provides support for the emulation of applications with certain characteristics. The main requirement is actually that the application interacts with the underlying operating system or a middleware in some way. Since one of PiCSE's requirements is that application code is unmodified, the framework exploits the application's calls to the operating system or to the middleware as an opportunity to redirect the application. Ironically, simple applications which have no interaction with libraries or the underlying operating system are also not suitable, as there is no easy way of controlling their execution without modifying the application code itself. Technically PiCSE can support such an application provided that it will eventually exit, but in any case, an application such as this would not actually be doing anything "interesting", since it has no inputs or outputs.

There are some limitations to the approach that has been implemented. In order to maintain fine-grained control over the execution of an application, it was chosen to not use an off the shelf virtual-machine based approach but instead to modify libraries that the emulated applications have been built on so that they can interact with the PiCSE simulation environment. If an application has been built upon a library that has not yet been extended, then that application cannot be emulated with an experiment.

At present, no threading libraries such as the POSIX library have been ported for inclusion in the PiCSE architecture and as a consequence, emulation of multi-threaded applications is not yet supported. In reimplementing part of a library such as the POSIX library, a mapping would have to be provided between the scheduling functionality of the library to the corresponding functionality within the PiCSE core architecture. The ported library would have to provide an API that supported the scheduling, execution, cessation and interruption of applications that would be built upon that library. However, fine-grained control of the individual threads would not necessarily be required for PiCSE to be able to support multi-threaded applications.

The same level of application control that is required for multi-threaded applications would be required when executing distributed simulation experiments, albeit at a more course

level of control. In addition to this, the distributed nature of the simulation would require the inclusion of a coordinating component at the control level that provided and managed the functionality to identify node simulators competing and accessing conflicting resources with an experiment. In addition, this component would have to provide the functionality to potentially resolve any arising conflicts, typically using a rollback feature whereby a remote node simulator could roll back its simulation to a previous state. On the node simulator side, this would require the implementation of a component that could store a series of rollback states, which would be subsequently released by the coordinating simulator when no conflicts have been identified. There is extensive literature (Fujimoto 00) on the suitability of extending DEVS-based simulators to distributed environments, a primary reason why a DEVS-oriented approach was originally adopted, however this functionality and components have not been built into the current implementation of the PiCSE simulation framework.

3.8.3 Supporting Multiple Applications

PiCSE is based upon the discrete event simulator paradigm, but application emulation and execution does not lend itself naturally to this model. A discrete event simulator executes as quickly as possible, and individual events take zero 'simulated time' to execute. Applications, both real and simulated, do however take time to execute, and it is difficult to control the execution of application instructions in a controlled manner, so that the execution of discrete model events and the execution of the application's instructions are correctly ordered. In fact it is very difficult without controlling the individual execution of the instructions forming the application and having knowledge of the time taken to complete each instruction. This approach is known as "binary translation" and is very cumbersome. An alternative software-based approach is possible in cases where the application source code is available. The ability to support multiple applications, developed independently contributes to meeting requirements R2 and R3, the two requirements relating to the support for the flexible integration of emulated software elements. This approach is now discussed in section 3.8.3.

Discretising an Application

One of PiCSE's main requirements is that no modifications are made to emulated applications and therefore all control of the application's execution must occur outside of the application itself. It is possible to discrete-ise the execution of the application. The execution of a series of instructions comprising a part of an application would occur at a discrete point in simulated time, and would therefore take zero time to complete. PiCSE makes this assumption and thus allows applications to be adapted transparently to an execution cycle that is suitable for execution within a discrete event simulator.

In practise this means that an application starts or resumes its execution whenever an event occurs that should normally trigger the application's execution. When the application reaches the end of its current execution cycle, for example, it pauses its execution while waiting for another event or perhaps invokes the system call `sleep(int seconds)`, the processing of the event list and the advance of other simulated events then resumes. The PiCSE framework provides functionality to control the execution, pausing, and resumption of emulated applications. It does this in a transparent manner from the application's perspective. However, this comes at the expense of accurate modelling of the time taken for the program to execute. PiCSE makes attempts to alleviate this problem, allowing a "pause" to be introduced transparently into an application's execution, thus slowing down the emulated execution of the application. The result is that the net simulated-time taken for a series of instructions to be executed can be modelled correctly, if that time taken to execute those instructions in real life can be measured. So if for example, a set of instructions has been measured to take 5ms to execute in real life, PiCSE will execute these instructions in 0ms (i.e., a discrete event), but can then model a pause of 5ms before the next instruction is executed.

The ability to approximate the total time taken to execute an aspect of a programs functionality combined with a similar level of fine-grained control of other time-related aspects of the domain such as hardware events and domain-specific events goes towards addressing the secondary challenge identified in chapter 1 of accurately modelling the timeliness requirements of pervasive computing application domains.

Executing Multiple Applications

A multi-threaded approach is implemented to support this feature of PiCSE. Applications are executed within their own thread, whose execution is controlled from within the simulator. A master-slave thread implementation controls the switching between threads. The master thread manages the processing of the event list, thus advancing the overall simulation, and also manages the execution of the slaves and the applications.

The PiCSE event list and event scheduling mechanism is modified to allow for different event types. These types include `Model_Events`, `Thread_Begin` events and `Thread_Switch` events. All event type occurrences are treated discretely, i.e. they take no simulated time to occur. When an event of type `Thread_Switch` occurs, the master thread restarts the appropriate slave thread (which is the application), and immediately relinquishes its own thread of execution. It is the application's associated `ExecutionEnvironment` instance that ultimately relinquishes control back to the master thread.

Although there may be multiple applications (and corresponding threads) in the simulator, there is only ever a single thread being executed at any one time. The PiCSE_Core exploits the thread's capabilities merely to retain control of the execution of an application, in order to achieve a discrete execution of the application, and not to actually have multiple threads or applications executing concurrently. At present, only applications written in C++ (Stroustrup 98) have been emulated. The source code of the application is compiled into the simulator to make a single program. In addition to this, only single threaded applications are supported at present. In order to enable the emulation of multi-threaded applications, the pthread library will have to be partially ported, so that the underlying thread scheduling behaviour does not interfere with the threading mechanism of the PiCSE_Core.

3.9 Experimental Support

The well known Observer design pattern (Gamma 95) is supported within the PiCSE framework and is implemented by all of the abstractions that are provided to model the "hard" aspects of pervasive computing domains. This design pattern is exploited by the PiCSE

framework to provide a `Logger` abstraction that eases and supports the task of logging "events of interest" and results during the course of a simulation's execution.

Several derived abstractions such as `the Environmental_Layer_Logger`, `Entity_Layer_Logger` and `Object_Logger` are also provided, all of which provide logging functionality but for different sources of interest. Concrete instances of the `Logger` abstraction can subscribe to events scheduled by instances of the `Simulated` abstraction, such as `Sensor` and `Actuator` instantiations, and are notified when these events occur. A recording of the event and any additional parameters required is then written to a flat text file. Instances of the `Logger` abstraction can be additionally parametrised using the following two methods:

1. The method `::setPeriod(int period)` is used to request periodic logging of some aspect of a simulated domain, irrespective of when events of interest may occur.

2. The method `::setLoggingWindow (int time2start, int duration)` is used to define a period of interest within an experiment. This is useful for experiments in which there is a boot-strapping period and logging during this period is not required. A trigger can also be used to define the time at which to begin logging if it is not known at the time of instantiation.

3.10 Summary

This chapter introduced and derived the requirements that guided the design of the PiCSE framework. The architecture of the *PiCSE_Core*, the underlying infrastructure that supports all instantiations of the PiCSE framework, was explored. The main abstractions supported by PiCSE were then introduced. These were divided between abstractions that supported the modelling of the physical aspects of a pervasive computing scenario, and abstractions required to support the applications emulated within that scenario. A further examination was then provided of the framework's support for emulating applications that exist within the simulated application scenarios.

Chapter 4

Modelling the Key Pervasive Computing Components

This chapter describes the *PC_Abstraction* class category, which provides all abstractions required to create an instantiation of the PiCSE framework. This chapter begins by examining the relationships of the components and their role within a PiCSE instantiation. The components themselves, their flexibility and suitability for supporting the modelling of the components and their implementations are then addressed individually.

4.1 The PC_Abstraction class category

The *PC_Abstractions* class category provides a set of abstract classes that can be instantiated to meet the modelling requirements of the components identified to be common across pervasive computing scenarios. It is intended that developers instantiate classes from this category, or create new classes using inheritance where required. Dependencies between instances of these classes are then modelled to create more complex instantiations. These instantiations interact with a single instantiation of the *PiCSE_Core* to form an executable simulation. Alternatively, developers may create more specialised abstractions creating their own framework, that may itself be instantiated for a particular pervasive computing domain.

In relation to the logical architecture presented in chapter 3, instantiations of abstractions

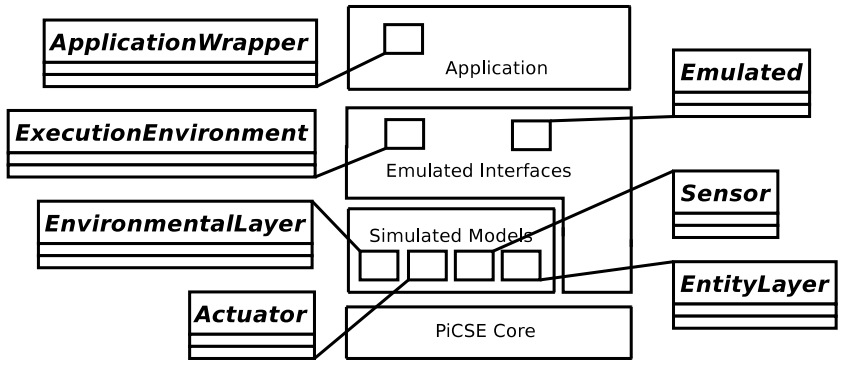

Figure 4.1: The location of the PC_Abstraction classes within the logical architecture.

from this class category are found in both the *Simulated_Models* and the *Emulated_Interfaces* layers, as shown in figure 4.1. The classes within this category provide the abstractions for the simulated models and also abstractions enabling the emulated applications to be run within instantiations of the framework. As such, this class category can be separated into two parts addressing separate requirements R1, R2, and R3 introduced in chapter 3.

- **R1** The framework should support the flexible simulation of the heterogenous hardware elements of pervasive computing scenarios. This is provided by creating customisable abstractions representing physical aspects of the scenarios such as the environment, as well as hardware devices such as sensors and actuators.

- **R2** The framework should support the emulation of real-code applications developed for pervasive computing. These applications may be built upon middleware that should also be supported.

- **R3** The framework should mediate the flexible interaction of any simulated and emulated elements forming the simulation of a pervasive computing application.

Five abstractions are described in this chapter and these can be broken into three logical categories: objects, the environment, and application environments. Both object- and environment-related abstractions are located within the *Simulated_Models* Layer in the logical architecture, and the implementation of these addresses requirement R1. The application

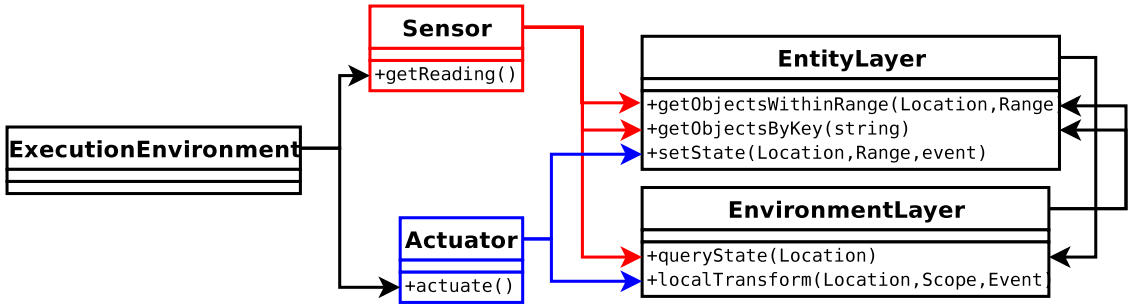

Figure 4.2: Objects instantiated from the PC_Abstraction class category are pre-defined to interact using certain methods.

environment abstractions support instantiations from the *Emulated_Interfaces* Layer and the implementation of these address requirement R2.

Figure 4.2 shows the typical simplified interaction of instantiations of these components. As outlined in chapter 3, the support for these instantiations is provided by the underlying *PiCSE_Core* class category. In chapter 2, the common interaction patterns of these components were identified, and the design and implementation of the abstractions reflect the requirements of meeting these interaction patterns. In this chapter, the features of note, the commonalities and the interaction patterns identified in chapter 2 are addressed as they apply to each of the five abstractions.

4.2 ExecutionEnvironments

The `ExecutionEnvironment` abstraction supports the emulation of the environment in which an application would normally execute. The abstraction supports applications that are built on API's of the underlying platform, and also applications built on middleware instances that abstract these underlying platforms. The abstraction consists of two parts. It supports the building of libraries that provide access to the modelled hardware devices within a PiCSE instantiation. Secondly, the abstraction supports the creation of an API, upon which applications can be built, allowing the applications to interact seamlessly with the modelled devices.

The abstraction also supports the modelling of the networking functionality of an op-

erating system or middleware platform, through the networking interface provided by the `Net_Interface` abstraction. The following requirements exist amongst all execution environments.

- An `ExecutionEnvironment` abstraction can interact with simulated hardware devices, in the form of instantiations of the `Sensor`, `Actuator` and `Object` abstractions. The abstraction must also support the interaction of the `ExecutionEnvironment` with the underlying PiCSE_Core.

- The `ExecutionEnvironment` abstraction must provide the interfaces required by the application, so that the code of applications can be preserved without change. This allows seamless emulation of the application behaviour.

- The role of any `ExecutionEnvironment` instantiation is to mediate the interaction of these two interfaces.

Split-Level API Implementation

The mediation between the different interfaces is achieved logically using the split-level API approach that was described in 3.8.1 of chapter 3. The `ExecutionEnvironment` abstraction supports the instantiation of the middle layer, which is termed the Replaceable Emulation Unit. This middle layer binds method invocations from the application, through the API to the corresponding methods within the PiCSE instantiation. One method of doing this is to rewrite partially the libraries on which the application has been developed. Using this methodology, the PiCSE instantiation, and all simulated models including the hardware devices, the environment, and the network, are all transparent to the application.

The Replaceable Emulation Unit only has to be written once to enable the emulation of a family of applications. For example, one instantiation of the `ExecutionEnvironment` abstraction is required to create a `TinyOS_ExecutionEnvironment` that would enable the emulation of all TinyOS applications. Similarly, one instantiation would exist for the Context Toolkit middleware and so on.

The mediation of the two interfaces is supported within the `ExecutionEnvironment` abstraction by another abstraction of a file, called `PFile`. The `ExecutionEnvironment` abstraction supports a collection of these, that can be addressed by a key (analagous to a filesystem path) providing an abstract model of a filesystem. Both applications and hardware devices can read and write to these `PFile` instantiations, that can act as a logical buffer between the higher level (applications) and the lower level (hardware devices).

Application Interaction

Applications are supported in the `ExecutionEnvironment` abstraction through the `ApplicationWrapper` class. The wrapper class allows applications to be associated with a single `ExecutionEnvironment`, and allows the `ExecutionEnvironment` to schedule the initialisation of the application using the `::scheduleExecution(long int t)` method.

Hardware Interaction

Models of hardware devices are integrated into the `ExecutionEnvironment` abstraction using their `Emulated` interface. This interface will be introduced in more detail in the following sections describing the `Sensor` and `Actuator` abstractions, but very briefly, it provides an interface allowing an `ExecutionEnvironment` instance to interact with an instance of a simulated hardware device. From the `ExecutionEnvironment` perspective, each `Emulated` interface can be hooked into a `PFile` instance. The `Emulated` interface of a sensor can push a sensor reading through the `PFile` interface, where it can be read by an application from that emulated filesystem. A similar data flow occurs in the opposite direction for actuators. Applications can write commands to the `PFile` interface whereby they are pushed to the `Actuator` instantiation and the resultant simulated actuation is performed.

Network Simulator Interaction Typical pervasive computing applications may also communicate with their peers and/or external services using wireless or wired communication, or both. Excluding situations where middleware is used to provide abstractions, this is usually performed at the transport layer using known abstractions, such as

a 'socket'. The `ExecutionEnvironment` abstraction provides the `Net_Interface` interface, which supports this functionality. This interface allows messages to be sent to PiCSE's `Network_Simulation_Manager` which supports the simulated broadcasting and uni-casting of messages across wired and wireless networks.

Abstraction Limitations

Theoretically, PiCSE supports the emulation of large numbers of these instantiations. No upper limits are placed on the number of devices or applications that a single `ExecutionEnvironment` instantiation can support, and the PiCSE framework itself places no constraints on the number of `ExecutionEnvironment` instantiations. As described in chapter 3, however, each application runs within its own thread of control, so the overall number of applications may be constrained by the number of threads supported by the physical machine used to run the simulation.

4.3 Sensors

The `Sensor` abstraction is provided as a generic abstraction of all physical sensors that can capture physical phenomena. According to the Oxford English Dictionary (OED) (Soanes 05), a sensor:

> "is a device giving a signal for the detection or measurement of a physical property to which it responds."

There are two parts to this definition and both are captured within this abstraction. The first is that a sensor measures a physical phenomenon. Within the earth and ocean science communities, the term software sensor (Chen 98; Masson) has gained traction in recent years and is used to describe a system used to "compute an estimate of some quantity of interest, based on a mathematical model and other (faithful) measurements."

The PiCSE `Sensor` abstraction only captures devices that measure physical phenomenon, such as temperature, noise, etc, but does not provide any support for the computation or estimation of those physical phenomenon. This is instead captured in the `EnvironmentLayer`

described later in this chapter. The OED definition also captures that a sensor presents a signal, i.e., a reading, to the user of that sensor. Both of these interactions, between the sensor and physical phenomenon, and between the sensor and its user, are captured within the `Sensor` abstraction.

Within the broad definition of pervasive computing, there are a wide range of sensors with a range of properties that have to be met. Consider for example two different types of carbon dioxide sensors. The first, deployed in an environmental monitoring scenario could measure the ambient levels of CO_2 in the sensor's locality, and could potentially be on a sensor board interacting with an embedded operating system such as TinyOS. A second CO_2 sensor, deployed as part of an engine management system in a vehicle, would be controlled by an on-board computer, accessed across a controller area network (CAN)-bus and would return a value capturing the CO_2 emissions of the vehicle. These are two examples of sensors measuring the same phenomenon, but each having unique characteristics. In the first, the measured phenomenon is dependent on the location of the sensor within its environment. In the second, the measured phenomenon is independent of the location, but dependent on the attributes of a modelled object, a vehicle. The basic `Sensor` abstraction meets the requirement that a sensor can measure a phenomenon from a wide range of sources with varying characteristics.

4.3.1 Push and Pull models

An additional characteristic that is captured by the `Sensor` abstraction is whether the sensor is active or passive. An active sensor in effect measures or "pulls" its measurements from the phenomena that it is sensing. A passive sensor is driven by changes in the phenomena that it is measuring and measurements are effectively "pushed" onto the sensor device. An example is a photosensitive switch, that is activated whenever the level of light in its environment reaches a certain threshold.

4.3.2 Measured Phenomenon

The `Sensor` abstraction can support a diverse range of measurable phenomenon, which are classified as being either exteroceptive or proprioceptive. An exteroceptive sensor means that the sensor takes it readings from external stimuli, i.e. its environment. Proprioceptive sensors take their readings from objects to which they are physically attached. In the case of exteroceptive sensors, instantiations are parameterised to query an instantiation of either an `EnvironmentLayer` or an `EntityLayer`. Instantiations of proprioceptive sensors typically query instantiations of the class, or classes derived thereof. An example of a proprioceptive sensor would be a GPS sensor measuring its "own location" or the vehicle CO_2 sensor mentioned previously. A thermistor measuring the ambient temperature in the room would be bound to a `EnvironmentLayer` instance that models the temperature of that room.

4.3.3 Querying

Depending on the modelled phenomenon that a sensor measures, a reading is obtained using one of three native methods.

1. `EnvironmentLayer` instantiations are queried using the `EnvironmentLayer::queryState( Location l)` method. The sensors location is used as the default parameter.

2. `EntityLayer` instantiations are queried using the `EntityLayer::getObjectsWithinRange(Location l, double range)` method, which returns a group of `Objects` within a certain location.

3. `Object` instantiations are queried using a simple `Object::getReading()` method, that can return either a numerical or a more complex object type such as a location.

The returned value in all of these instances can be passed directly to the user of the sensor, in which case the sensor can be conceptualised as a user's or application's gateway to the physical environment. Alternatively, the value can be manipulated and processed to create a more realistic reading that is representative of an actual physical sensor.

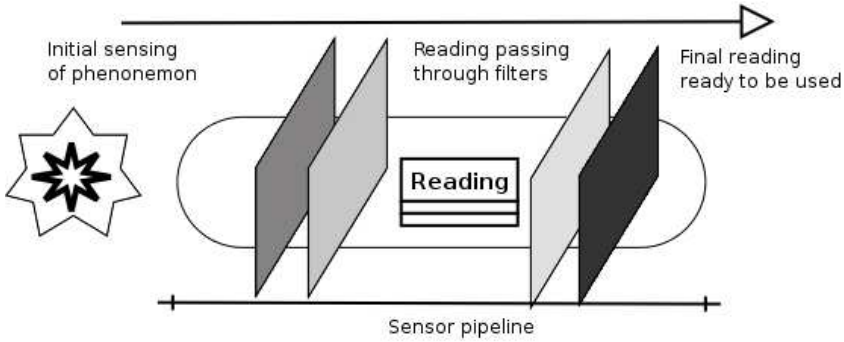

Figure 4.3: The process by which a reading passes through a pipeline formed of a combination of blocking and modifying filters

4.3.4 Characteristics of Individual Sensor Readings

Independently created sensors that measure the same phenomenon may have unique properties, such as different levels of accuracy and precision, that may affect the final reading that is presented to the user of that sensor. An additional property could be a threshold that a sensor may not be able to measure above or below. These functional characteristics have to be supported within the abstraction.

A flexible method for modelling these properties is to use a sensor pipeline, displayed in Fig 4.3. The sensor pipeline comprises of a combination of filters which may "modify" or "block" a measured phenomenon in some way. The initial measurement is made when the sensor retrieves data from the measured phenomenon, either an `EntityLayer`, an `EnvironmentLayer` or an `Object` instantiation. This initial *raw* value is then pushed onto the sensor pipeline where it passes through a series of filters. "Modifying filters" may update the value in some way by adding some error based on a normal error distribution for example. A "blocking filter" determines if it was physically possible to make the original reading, and may take the location of the sensor and the distance to the sensed phenomenon into account. All filters may themselves be dependent on other modelled components within a simulation, for example, be that `Object` or `Layer` instantiations. By combining many of these filters into a single conceptual pipeline, through which all sensor measurements must pass, it is possible to provide a flexible and potentially, accurate model of a sensor.

4.3.5 External Interfaces

The remaining part of the Sensor abstraction is the delivery or the presentation of the reading to the application. In PiCSE terms, this is mediated by an instantiation of the `ExecutionEnvironment` abstraction that plays the role of either a device driver for the sensor, or some middleware that abstracts that functionality. The role of modelling the interaction is split between the `Sensor` and the `Emulated` abstraction. The instantiation of the sensor is responsible for the formatting of the filtered data into a form understood by the `ExecutionEnvironment`.

The `Emulated` abstraction, not previously introduced, is used to assist in the modelling of the interaction between the object and the `ExecutionEnvironment`. Its role is to manage the interaction of the sensor with an instance of an `ExecutionEnvironment`, in a way that enables the accurate emulation of the interface. For example, an `ExecutionEnvironment` instantiation may represent a Sensor as part of the file system. In this case, the Emulated instantiation pushes the sensor readings, as they occur, to the appropriate part of that modelled file system.

4.4 Actuators

Traditionally, actuators would have been used within closed or embedded environments, such as manufacturing or traffic control systems. Non-functional properties, such as safety and security, had to be considered (and still are), and these closed systems were best suited to meeting these requirements. In recent years, Wireless Sensor and Actuator Networks (WSANs) have become a well established paradigm, and projects such as WiSeNts[1], and now CONET[2] are exploring the interaction between embedded systems, pervasive computing, and wireless sensor networks.

The use of actuators represents a flow of effect in the opposite direction to that of sensors: sensors, in the shape of hardware or software measure some aspect of their environment and applications consume sensor data to produce some meaningful results; whilst actuators enable applications to affect the environment. Therefore, the integration of actuators in

[1] www.embedded-wisents.org
[2] www.cooperating-objects.eu

pervasive computing completes the loop of the interaction between the application and the real world. (Verdone 07) clarify the definition of an actuator as "a device able to manipulate the environment rather than observe it". It is this "manipulation" of the environment that the PiCSE `Actuator` abstraction must capture.

4.4.1 ExecutionEnvironment Interaction

The implementation of the actuator abstraction is similar to that of the Sensor abstraction. There is however no push and pull distinction in actuators. All actuators either act independently, or more commonly are under the control of an application. Again, the interaction of the emulated application and the model of the simulated actuator is mediated by a combination of an `ExecutionEnvironment` instantiation and an `Emulated` Instantiation. In this scenario, the role of the `Emulated` instantiation is to parse or interpret any command that the application may issue, and invoke the appropriate method within the `Actuator` instantiation. This is analogous to the "formatting" role played by the `Emulated` instantiation in the `Sensor` abstraction.

4.4.2 Modelling Actuator's Effects

Within pervasive computing, the environment is not just the static physical environment, such as the road or the atmosphere, but also includes objects within that environment, that may themselves be a part of the scenario. The Actuator abstraction therefore can interact with the three PiCSE abstractions, `EnvironmentLayer`, `EntityLayer` and `Object` that are provided to model this environment.

There are two methods of modelling actuation on the environment within PiCSE, and they are distinguished by where the knowledge of the effect of an actuator lies. The first method places the responsibility of modelling the effects within the component that has been actuated upon. In this case, the actuator schedules an event within the component, and when the event occurs, potentially immediately, the component consumes the event and updates some attribute to reflect the effect of the actuation.

Alternatively, the actuator itself may maintain a reference to the component on which it

actuates, and updates the state itself. Using the first technique, the affected component must have an understanding of the capabilities of the actuator. Using the second technique, an actuator must have knowledge of the modelling aspect of the environment that it is updating. The main abstractions, `EnvironmentLayer`, `EntityLayer` and `Object`, present an API that exposes their state allowing `Actuator` instantiations to "actuate" upon them.

As introduced in the `Sensor` abstraction, the use of a "filter" pipeline is also supported, allowing the modelling of more complex actuation events, that are perhaps dependent on more than one aspect of the modelled environment. In the actuator pipeline, the filters can be used to determine whether the actuation can actually occur and what its effects will be on the environment.

4.5 EnvironmentLayers

The simplest most abstract concept within the PiCSE framework is that of the physical phenomenon. A physical phenomenon is a tangible aspect of an environment within a pervasive computing domain and is captured within the `EnvironmentLayer` abstraction. By tangible, it is meant that the physical phenomenon is both detectable and can also be manipulated. In the context of pervasive computing, that means that it can be sensed and actuated upon by hardware devices. An `EnvironmentLayer` abstraction supports the modelling of a single phenomenon. `EnvironmentLayer` instantiations can also be grouped and they then collectively form part of a simulated model of the environment. A group of `EntityLayer` instantiations form the remainder of the modelled environment.

4.5.1 A Single Modelled Phenomenon

The most generalised representation of a physical phenomenon, representing some aspect of an environment, is that it has a certain state or value at a certain location at a certain time. The `EnvironmentLayer` abstraction uses location as the focal point of a grid-based approach to modelling that physical phenomenon. The `EnvironmentLayer` abstraction implements a three-dimensional grid of fixed size, that maps onto the scenario being modelled. In addition

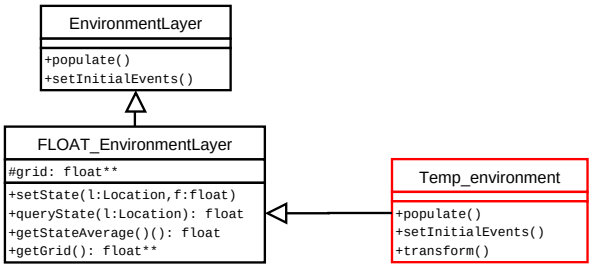

Figure 4.4: Temp_environment class

to the physical x, y and z dimensions of the space, each `EnvironmentLayer` instantiation is parametrised with the granularity of the grid within the layer. The granularity of a "space" within the grid can be arbitrarily large or small and offers an additional degree of flexibility. Within each discretised part of this grid, the value of modelled phenomenon is the same, as at all other points within the same discretised space.

A physical environment is of course a dynamic system, and the `EnvironmentLayer` abstraction provides an abstract method that can be implemented to update the phenomenon represented. The abstraction can be parameterised to evolve periodically, i.e. an abstract method, `::transform()`, that is equivalent to a state transition function, can be invoked at fixed time intervals. Alternatively, the transformation can be driven by events that occur in other `Layer` or `Object` instantiations. This is examined in detail later.

From a concrete class point of view, an `EnvironmentLayer` object can represent any physical phenomenon from the definition above. Figure 4.4 shows the class hierarchy for a Temperature EnvironmentLayer instance. The abstract populate() method is defined to set the initial environment temperature values and the abstract transform() method defines to specify how those values change over time.

The abstract implementation of `EnvironmentLayer` affords developers the freedom to tailor the layer to their requirements and the parameters of freedom within an instantiation include the size of the modelled space, its granularity, its state and a state transition function that is invoked to update the state of the physical phenomenon at a particular time.

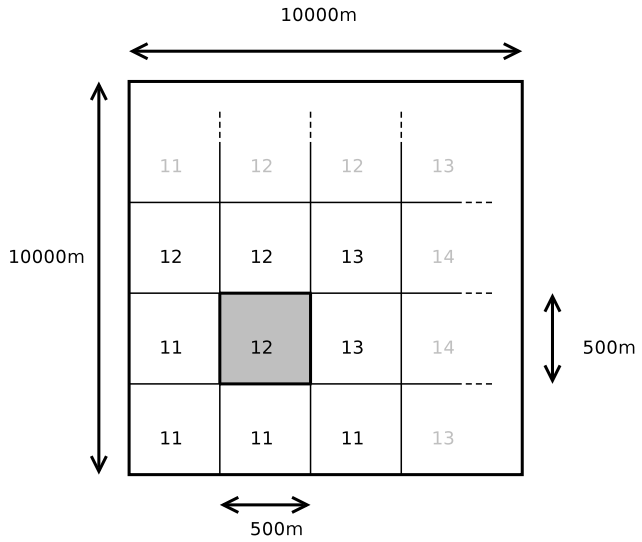

Figure 4.5: A PrecipitationLayer object models rainfall within the simulated environment.

The Modelling of Continuous Phenomenon

Whilst some phenomenon lend themselves to coarse-grained modelling, other more continuous phenomenon such as natural or physical phenomenon are described by mathematical formulae. In these cases, an inherited instance of the EnvironmentaLayer class instance should provide a method implementing this formula, which can then be invoked to determine the state of the phenomenon at a particular point in time. This approach is provided as an alternative to the "discretised" approach presented above.

Environmental Monitoring Scenario

Consider an environmental monitoring scenario. A set of sensors are physically deployed across a large geographic space and communicate wirelessly to aggregate sensor information that has measured the precipitation in an environment. In modelling this scenario, an `EnvironmentLayer` object, PrecipitationLayer can be instantiated to model the amount of rainfall, a physical phenomenon.

By way of example, figure 4.5 , shows the `PrecipitationLayer` model, representing an area of 10km*10km with a discrete granularity of 500 metres. The state modelled is the

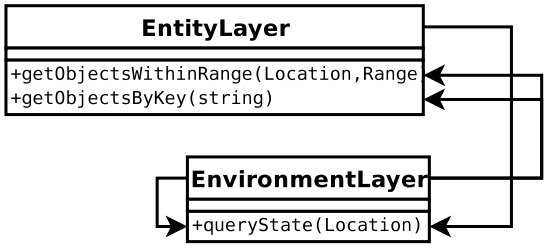

Figure 4.6: EnvironmentLayer instantiations can interact directly with other Layers

amount of rainfall, measured in millimetres. Rainfall is a coarse-grained phenomenon[3] but a smaller granularity can be used if more accurate modelling is required. A transition function is defined to capture the amount of rainfall dependent on the time of day. Separate meteorological conditions can be captured within other `Layer` instantiations.

By varying the granularity, and the corresponding `::transform()` method, a wide range of phenomenon can be modelled. `EnvironmentLayer` objects can be tailored to model other simple physical phenomenon such as the topology of the ground, the presence of buildings or roads, and the levels of light, noise or temperature at a particular location within the environment.

4.5.2 A Location-Based API

The `EnvironmentLayer` abstraction is used to model physical phenomenon. The role of sensing and actuating is important within the definition of the `EnvironmentLayer` abstraction, since it is only sensors and actuators that can interact with the modelled environment in PiCSE instantiations. Considering this, the `EnvironmentLayer` abstraction has to support that interaction and it can be said that this interaction is the driving force behind the location-based API of the abstraction.

Sensing

When an `EnvironmentLayer::queryState( Location )` method is invoked, the `EnvironmentLayer` object maps the location parameter to a region within the grid. The state

[3] Rainfall does not change in quantitative terms significantly from metre to metre, so it possible to use a large regional granularity.

of the `EnvironmentLayer` object at that region is then returned. A sensor can 'sense' the state of the modelled phenomenon by invoking the `::queryState(Sensor::getLocation())` method using its own location as the parameter. Two sensors whose respective locations both map to the same region within an `EnvironmentLayer`, as determined by the `EnvironmentLayer` granularity, will both sense the same value of the modelled phenomenon. The `::queryState(Location)` method abstracts the variable granularity and grid sizes of different instantiations, thus simplifying the interaction.

Actuating

The `EnvironmentLayer` provides two methods, one virtual and one abstract, to support the interaction of `Actuator` and the `EnvironmentLayer` instantiations. The first method `::setState(Location, state)` provides a simplistic 'set' corresponding to the 'get' method `::queryState(Location)`. This method may be invoked by the actuator, and from the perspective of an EnvironmentLayer instantiation, it is the actuator that is performing the action.

Alternatively, the `EnvironmentLayer` class provides an abstract method `::localTransform(Location source, double scope, int event=0)` that can also be invoked by an actuator. In this instance, the `EnvironmentLayer` object is required to implement a localised version of its own `::transform()` method, that takes into account the source, scope and type of event that the actuator has generated.

The first method `::setState(Location, double)` is limited, since it does not take into account the potential scope of an actuator's actions. Furthermore, the actuator is required to understand the semantics of the behaviour of the `EnvironmentLayer`, so that it can update the state to an appropriate value. For this reason, the second method is preferred when modelling the effects of an actuator's actions. However, this method has its own limitations as the technical capabilities of the actuator may not be known, and providing alternative means of interaction between `Actuator` objects and `EnvironmentLayer` objects helps meet the flexibility requirement. Furthermore, providing a localised version of the `::transform()` method improves scalability, as it reduces the time and computation required in modelling the interactions between actuators and the environment.

4.5.3 `EnvironmentLayer` Complexity

Using logical dependencies, a large-scale representation of a range of environments can be modelled by instantiating multiple `EnvironmentLayer` objects and creating dependencies between them.

The `EnvironmentLayer` abstraction supports the modelling of more complex phenomenon through the framework's underlying support for logical dependencies. An `EnvironmentLayer` instantiation can implement its state and `::transform()` methods to be dependent on other instantiations in a simulation. For `EnvironmentLayer` instances, these are often alternative `EnvironmentLayer` objects or `EntityLayer` objects, although technically all simulated objects support the logical dependencies paradigm.

By way of an example, consider a simple extension to the environmental monitoring scenario just introduced. A new class `TemperatureLayer` is introduced that models another aspect of the physical environment, the temperature. More specifically a scientist wishes to capture the principle that increases in the temperature increase the likelihood of rainfall. This relationship is enabled and captured within the PiCSE framework as a logical dependency, i.e. a `PrecipitationLayer` instance is dependent upon a `TemperatureLayer` instance. In this extended scenario, the `PrecipitationLayer` instance, can subscribe to events from the `TemperatureLayer`, and can also use the `TemperatureLayer` interface to determine the temperature at certain locations.

The separation of the grid-based implementation from the API means that complex `EnvironmentLayer` instantiations can be built by treating other layers as simple state information, that is location dependent. Consequently, `EnvironmentLayer` instantiations can be logically collocated to create a more complex model of an environment, that is independent of the underlying implementation.

4.6 EntityLayers

The `EntityLayer` abstraction implements the second `Layer` abstraction, completing PiCSE's support for modelling of pervasive computing environments. Many different types of objects

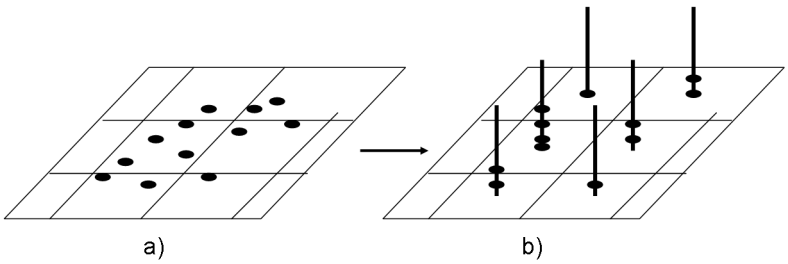

Figure 4.7: `Object` instances within close proximity, defined by the grid's discretised space, are bundled into a single list

can be found in pervasive computing, such as vehicles in intelligent transportation system (ITS) scenarios, people in smart space scenarios, and RFID tags in logistics. In almost all cases, these objects themselves form part of the environment in which they exist. They interact with each other and can affect the physical aspects of the environment that are measured by sensors, as well as being detectable by sensors themselves. The `EntityLayer` abstraction provides a model that supports the modelling of these objects within a physical space.

Similar to the `EnvironmentLayer` abstraction, the location of these objects is central to the role within pervasive computing environments and this is reflected in the implementation of this abstraction. The same underlying model of a discretised grid of physical space exists, but each of these discrete spaces no longer represents a physical phenomenon, but contains a list of `Object` instances within the discretised space, as shown in figure 4.7. All `Object` instances within a particular bounded region of an `EntityLayer` grid are in close proximity in the simulated space.

An alternative key-based approach allows groups of `Object` instantiations to be referenced by an identifier known to all. This approach is suitable for modelling scenarios where the absolute location of the `Object` instances is not as important, but some other non-location based identifier may be used. For example, referencing `Sensor` instantiations that are active, or updating the location of all `Object` instances representing people inside a building. In these scenarios, a key "ACTIVE" would be used to obtain all `Object` instances, and in the second scenario, the identifier of the building would be used to reference the person `Object` instances.

As noted in chapter 1, supporting the modelling of large-scale application domains is an important secondary challenge that must be met by PiCSE's architecture. The effective location management of simulated objects plays a large role in developing efficient simulations that support large scale scenarios. The use of a location-based referencing mechanism for modelling and management of physical components enables large-scale simulations by exploiting the location information to bound the amount of interactions that can occur between simulated models.

All `Object` instances, whether they are `Sensor`, `Actuator`, or domain specific `Object` instances, implement their own `::transform()` method, which may be invoked periodically or alternatively event driven. The `EntityLayer` abstraction allows all `Object` instances of a certain type to be treated as a single `Object`, and an instantiation of the abstraction can be parameterised to transform all objects independently or as a unit.

4.6.1 Location based Interaction

Similar to the `EnvironmentLayer` abstraction, `EntityLayer` instantiations can potentially interact with instantiations of `Sensor`s, `Actuator`s, other domain specific `Objects`, and other `Layer` instantiations. The `EntityLayer` abstraction provides two APIs based on the underlying modelling mechanisms mentioned in section 4.6, which are a location- and a key-based API.

Querying

Two methods are provided for retrieving collections of objects based on their location.

```
GROUP_OF_OBJECTS* getObjectsWithinRange(Location source, long double
range);

GROUP_OF_OBJECTS* getObjectsWithinBoundingRectangle(Location,
Location);
```

The first method, `::getObjectsWithinRange(...)` returns a list of `Object` pointers, which are within a proximity of length `range` of the location `source`. Figure 4.8 demonstrates

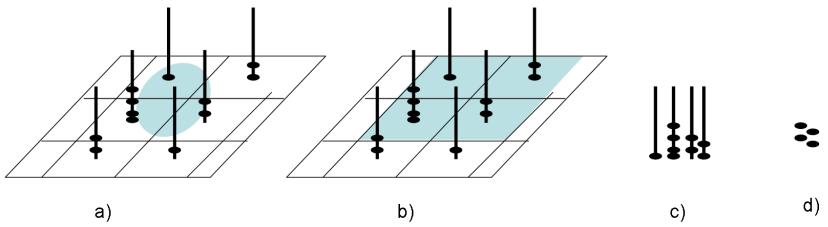

Figure 4.8: Calculating the Object instances within a certain range in EntityLayer instantiations

the process by which the nodes within range are calculated. The `EntityLayer` uses the location parameterised, and the range to calculate the discretised spaces that overlap with the parameterised range. Objects within these discretised spaces are then checked for proximity to the parametrised location. Finally, the list of `Objects` that are calculated to be within the range are returned. A similar method `::getObjectsWithinBoundingRectangle(...)` returns a list of `Object` pointers that are constrained by the two parameterised locations.

The key-based approach to querying is supported with the `::getObjectsByKey()` method which uses the key to index a hash-map of `Object` pointers, and then returns a list of `Object` pointers.

```
GROUP_OF_OBJECTS* getObjectsByKey(string key);
```

Updating

Updating the `EnvironmentLayer` abstraction is also supported using either the location- or key-based methods. An `EntityLayer` abstraction supports the modelling of objects within their environment, but not the actual objects themselves. So any event, either updating or actuation that is to be performed on a group of objects, i.e. an `EntityLayer` instantiation is performed on all objects individually. This mapping to the individual `Object` instances, is performed by the abstraction transparently.

The method `::setState(location, range, event)` applies an update to all `Object` instances, within a determined range. The method `::setState(location, location, event)` applies an event to all Object instances within a rectangle, bounded by the two parameterised

locations. Finally, the `::setState( key, event)` method applies an event to all `Object` instances indexed by that key. In the case of each of these methods, the abstraction, using the query methods outlined, determines the list of objects to which the event pertains, and then schedules the event to occur in 0 seconds in each individual `Object` instance. In the event, that two or more events are scheduled to execute at the same time step, the events are processed in the order that they were scheduled, but all at the same simulated time.

Mobility

Mobility of `Object` instances are handled transparently. As an `Object` updates its location, any `EntityLayer` instantiations that reference that `Object` are notified, and in the event that the `Object` has moved from one discretised part of the grid to another, the instantiation is updated using the `::updateObjectsGridLocation(Object*)` method.

4.7 PiCSE as a Combinatorial Framework

The PiCSE framework is intrinsically flexible as its architecture provides generic abstractions that can be instantiated to model the common components of pervasive computing scenarios. However, clearly not all components of all domains can be anticipated in advance and the *Abstract_Interfaces* class category provides classes and interfaces that allow developers to create models to meet their own domain modelling requirements. In addition to enabling the creation of new abstractions, the framework should support the capturing of logical and physical relationships between instantiations of the PiCSE abstractions, allowing the creation of models that are beyond the instantiation of a single abstraction. Both physical and logical dependencies are captured within the framework and supported by the class categories. Supporting the free interaction, combination and inter-dependence of simulated hardware and software elements within the framework is a key step towards addressing the challenge identified in chapter 1 as flexible heterogeneity and is also requirement R3 with respect to the PiCSE framework.

4.7.1 Physical Dependencies

The framework captures physical dependencies that occur regularly whilst creating realistic models. The main physical dependency concerns the location and collocation of objects. This recurring pattern captures the scenario where an object is being *"carried"* by another object, its *"owner"*. In this situation, a physical dependency is created, whereby the location of the *"carried"* object assumes the location of its owner. It is necessary to capture this relationship to allow instantiations of low-level object-based abstractions to be combined into more complex instantiations.

4.7.2 Logical Dependencies

The framework should capture the dependencies within a scenario whereby the state of an object is dependent on the state of another object. Take a typical environmental monitoring scenario, such as the forest-fire detection scenario described in (Yu 05). In this scenario, a wireless sensor network is deployed to detect forest fires. In the PiCSE framework, models of the environment are captured as sets of instantiations of the `EnvironmentLayer` abstraction. An instantiation of a scenario such as this one could include a layer representing a model of the forest and a layer representing the forest fire. A logical dependency exists here and is required to fully model this scenario, i.e. the state of the layer representing the forest fire is dependent on the layer representing the location or presence of trees.

The PiCSE framework supports the recurring pattern of logical dependencies, again supporting the creation of complex instantiations from combinations of simpler instantiations. Capturing this dependency requires that a change within a simulated model can initiate an update within a dependent simulated model. The PiCSE framework provides an event notification service, so that complex inter-dependent instantiations can be built if required. The nature of the dependency between two models can be captured as implementing either strict or loose causality.

4.7.3 Limiting the effects of updates

PiCSE supports the creation of complex simulated types, such as co-located `Objects` through the addition of an event publish-subscribe model. This can potentially lead to complex inter-dependencies between instantiations. Depending on the nature of these dependencies, and the accuracy of the simulation required, an actuator can cause events; that is events that occur as a result of an original actuation event.

PiCSE provides two approaches that attempt to mitigate the potential computational effect of simulating cascading events, although it does not fully prevent these errors at the human level. For example, it may be necessary for developers to specify cyclical dependencies between objects within a simulation, which if unchecked would result in an infinite number of cascading events. Typically this can happen when an update in one object would schedule an immediate update one or more other objects, which may ultimately include the original. To mitigate this, dependencies between layers can be specified as being either LOOSE or STRICT. A LOOSE dependencies do not necessarily result in an immediate update of state when an event occurs, whereas a STRICT dependency does.

1. **Loose Causality** Event notifications to dependent models are acknowledged and the `::transform()` method is invoked as usual at a scheduled time in the future. This is the default behaviour.

2. **Strict Causality** The reception of an event notification causes an immediate local invocation of the `::transform()` method.

In the event of loose causality being implemented, there exists a period of time between the periodic invocations of the `::transform()` method, indicated in figure 4.9 where the dependent model has not been updated to take into account the new data in the model upon which it is dependent. Implementing strict causality between dependent models however can cause cascading updates, whereby an update in one model forces another model to update, which can in turn force more layers to update and so on. Therefore, a trade off exists between maintaining the consistency of inter-dependent models and the effort required to maintain that consistency. The amount of simulation fidelity that may actually be lost by implementing the

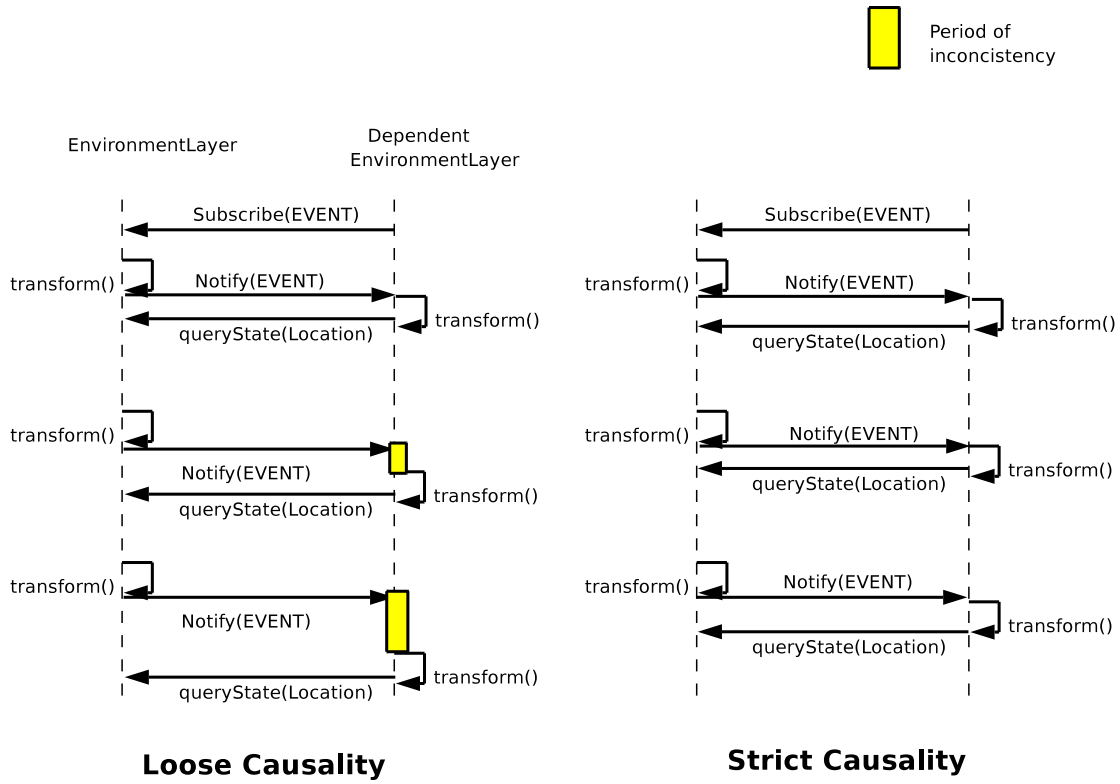

Figure 4.9: Logical dependencies can be captured using Loose or Strict Causality

loose causality is dependent on the modelled domain, and only the user can put a value on that fidelity. Aside from implementing strict causality as the alternative rigid approach, a lazy approach to causality would be implement a rollback function within the framework of the loose causality approach, whereby if a conflict was detected, an inconsistency could be reversed by rolling back the simulation to the point where the conflict arose and implementing a localised and immediate STRICT update.

Actuator instantiations can also exploit the location-based APIs of the `EnvironmentLayer` and `EntityLayer` abstractions to spatially limit the amount of actuation that is performed upon the environment. This means that computational time is not wasted evaluating the effects of an actuator on objects that are outside of its defined range. For example, when a speaker (actuator) makes a sound, it can specify a range that will determine what simulated elements will be actuated upon, which in this case means will hear the sound.

4.8 Requirements Analysis

The requirements determined for the PiCSE framework were determined at the outset of chapter 3, and they are once again presented and then examined to determine whether they have been satisfied by the architecture and design of the framework.

- **R1** The framework should support the flexible and heterogeneous simulation of the hardware elements of pervasive computing scenarios. This is be provided by creating customisable abstractions representing physical aspects of the scenarios such as the environment, as well as hardware devices such as sensors and actuators.

- **R2** The framework should support the emulation of real-code applications developed for pervasive computing. These applications may be built upon middleware that should also be supported.

- **R3** The framework should mediate the flexible interaction of any simulated and emulated elements forming the simulation of a pervasive computing application.

- **R4** The framework should support the creation of new abstractions of both hardware

and software elements, in order that future emerging application scenarios can be incorporated into the PiCSE framework.

- **R5** The framework should combine recurring elements of pervasive computing simulations into a set of reusable components thus reducing the amount of work required to instantiate any simulated scenarios.
- **R6** The framework should support network simulation for both wired and wireless networks.
- **R7** A framework that supports the creation of pervasive computing simulations must provide functionality to support the evaluation of those simulations.

Requirements R1, R2, and R3 The modelling of both hardware and software components is provided by the high-level classes, `Sensor`, `Actuation`, `EnvironmentLayer`, `EntityLayer`, `ApplicationWrapper` and `ExecutionEnvironment` and their respective inherited classes. PiCSE's support for modelling multiple simulated heterogeneous hardware and software elements and the framework's flexible implementation allows the potential interaction, combination and inter-dependence of these components without restriction. These three requirements have been met by the architecture, design and implementation of the framework.

Requirement R4 The support of the PiCSE framework's architecture to address the requirement of extensibility is met by two aspects of its design. An instantiation of the PiCSE framework can itself be a domain-specific sub-frameworks, composed of extensions of the software and hardware abstractions provided by the *PC_Abstractions* class category. Additionally, the *Abstract_Interfaces* class category provides a set of interfaces that allow the development of new abstractions that are not currently modelled within the existing framework implementation. These extensions can interact and interoperate with the existing *PiCSE_Core* and the existing abstractions. Additionally, the split-level design of the framework's emulation component allows new emulation application APIs to be added when in the future when new platforms or middlewares emerge. These functionalities and design features meet the framework's extensibility requirement, R4.

Requirement R5 The *PiCSE_Core* framework engine comprises of a set of components that are common to all PiCSE simulations. This reusable engine, and the identifying of recurring software patterns that are captured within the classes for the hardware and software elements, and their inter-dependencies meets the framework's reusability requirement, R5.

Requirement R6 The PiCSE framework provides two feature that address the aspect of network simulation within pervasive computing scenarios. The `Net_`Interface class and the underlying Network Manager component provide a rudimentary API for the simulation of network behaviour. This requirement, R6, is partially met.

Requirement R7 The PiCSE framework's provides a flexible API that allows a developer to specify time windows, where the the behaviour and state of objects of interest within a simulation is recorded. This requirement, R7, is partially met.

4.9 Conclusion

Overall, the requirements identified in chapter 3 have been satisfied. Requirements R1, R2, R3, R4, and R5 which address the core challenges identified in chapter 1 have been fully met in the design of the architecture and implementation of the PiCSE framework. The additional functional requirements R6 and R7 have been partially satisfied. It was decided not to incorporate a full network simulator into the framework's design as there are many existing well-established network simulators already in existence. Rather than compete with these simulators, the framework provides an open location-based API allowing all elements to be queried so that the integration of the framework with a third-party network simulator might be achieved in the future. With respect to the experimental support, there are no widely accepted standards for recording the output of simulated experiments in the pervasive computing domain so this feature has not been implemented. In-line analysis and debugging of pervasive computing application scenarios was also not implemented, and as such this requirement is only partially met.

Chapter 5

Evaluation

5.1 Introduction

The PiCSE framework has identified common characteristics of a set of recurring components, and implemented these in a set of class abstractions that can be customised to a wide array of pervasive computing scenarios. Instantiations of these abstractions can be freely composed into more complex instantiations, demonstrating the flexibility of the framework approach. This is supported by the abstraction's interfaces, that support the structured composition of these complex instantiations, whilst implicitly encapsulating recurring interaction patterns within the domain. This chapter presents three scenarios that demonstrate the fulfilment of the framework's requirements as they were outlined in the beginning of chapter 3. Both the simulation and emulation aspects of the framework are evaluated in three diverse independent application domains that highlight the diversity of pervasive computing scenarios.

5.2 Experimental Methodology

When evaluating a framework based architecture, the typical approach is to instantiate three distinctive yet related scenarios that exercise the ability of the framework to be flexibly adapted to those scenarios. In addition to this requirement of a framework, further requirements were determined for the PiCSE framework at the outset of chapter 3, and they are

once again presented.

- **R1** The framework should support the flexible and heterogeneous simulation of the hardware elements of pervasive computing scenarios. This is be provided by creating customisable abstractions representing physical aspects of the scenarios such as the environment, as well as hardware devices such as sensors and actuators.
- **R2** The framework should support the emulation of real-code applications developed for pervasive computing. These applications may be built upon middleware that should also be supported.
- **R3** The framework should mediate the flexible interaction of any simulated and emulated elements forming the simulation of a pervasive computing application.
- **R4** The framework should support the creation of new abstractions of both hardware and software elements, in order that future emerging application scenarios can be incorporated into the PiCSE framework.
- **R5** The framework should combine recurring elements of pervasive computing simulations into a set of reusable components thus reducing the amount of work required to instantiate any simulated scenarios.
- **R6** The framework should support network simulation for both wired and wireless networks.
- **R7** A framework that supports the creation of pervasive computing simulations must provide functionality to support the evaluation of those simulations.

The methodology adopted in this evaluation is to extend the subjective approach of framework instantiation to measure, quantitatively where possible, the success of PiCSE framework in addressing the additional requirements identified. By evaluating the scenarios individually and then subsequently drawing conclusions and insights across all three scenarios , a more rigorous evaluation is achieved. The metrics that have been identified to validate these requirements are outlined in table 5.1.

Requirement	Metrics
R1	The number of diverse instances of all hardware components
R2	The number of instances of emulated components, both applications and middlewares.
R3	The number of instances of component inter-relationships, which is defined as an interaction between two seperately created instances of PiCSE abstractions.
R4	The number of instances of inherited reusable abstractions. Measure the proportion of the simulation that is comprised of re-usable domain specific elements.
R5	The proportion of the simulation that is provided by the PiCSE frameworks core classes.
R6	Demonstration of network functionality in application scenarios.
R7	Demonstration of evidence of evaluation support functionality.

Table 5.1: Requirement evaluation metrics

These metrics are applied across three distinct pervasive computing scenarios. These scenarios, a *Steam Emulation* scenario, a *Car Hardware* scenario and an *Intelligent Transportation Systems* scenario are reflective of common application domains actively being considered within the pervasive community at this time and have recurring characteristics that will exercise the flexibility, reusability and extensibility of the PiCSE framework. These characteristics include

- Complex models comprising of many distinctive hardware and environmental components.
- Multiple application types written both natively and on middleware platforms.
- Complex interaction between hardware, software and environmental factors for the purpose of domain-driven objectives such as coordinated behaviour, environmental detection and distributed information gathering.

The three scenarios are presented in the following format. A high level description of the scenario is presented, in which its characteristics and key components are drawn out. This is then followed by a description of how PiCSE's abstractions provide the supporting functionality required to model these components and their interaction. The execution of the resulting

modelled scenario is discussed and any domain insights are highlighted. Finally, the evaluation measures how effective the instantiated scenario exercises the framework's requirements.

5.2.1 The Scenarios

The *STEAM Emulation* scenario models the interaction between multiple mobile pervasive computing applications, running on simulated PDA devices, that communicate using event-based middleware. This scenario captures the effects of mobility in the communication between the two applications, through the emulation of the event middleware, and its integration into the network simulator.

The *Car Hardware* Scenario is characterised by the modelling of the realistic integration of simulated and emulated components. The realistic modelling of the hardware interface is central to the effective emulation of the application, and this scenario exercises PiCSE's support for this requirement.

The third and final scenario presented is an *Intelligent Transportation Systems* scenario. A simulation of a smart Dublin traffic environment, this scenario models a higher level of complexity within the simulated aspects of the scenario including thousands of interacting vehicles, roads and smart traffic lights. This large-scale scenario captures a large degree of both physical and logical dependencies between a range of instantiations of PiCSE abstractions.

5.3 The STEAM Emulation Scenario

The first instantiation of the PiCSE framework models the emulation of two applications that communicate using event-based middleware. The STEAM middleware, described in detail by (Meier 03), enables event-based communication for collaborative applications in distributed environments. The middleware provides several underlying services abstracting from the underlying communication medium. An announcement service periodically broadcasts a list of available event types that may be subscribed to. A discovery service is monitoring the environment for these announcements. An application may use STEAM's production service to produce and broadcast these events, whilst a subscription service allows these events to

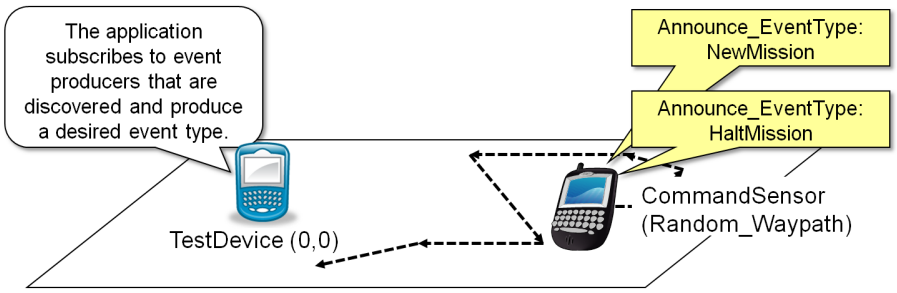

Figure 5.1: CommandSensor announces event types periodically

be subscribed to, and provides filtering functionality based on three criteria: event type, content and proximity. Combinations of these filters allow a fine-grain of control over the ultimate delivery of these events to the application built upon the middleware. All services are available on each STEAM middleware instance, but may not necessarily be utilised; for example, some instances may be event producers only, whilst others may be both event producers and subscribers.

In this scenario, depicted in figures 5.1 and 5.2, two applications running on handheld devices, one static and one mobile, use the STEAM middleware to exchange messages. The mobile device, `CommandSensor`, produces two types of message, "newMission" and "haltMission" every seven seconds, and moves around the physical environment following a random path. The static device, `TestDevice` subscribes to events of type "newMission", and upon receiving an event, an event handler is triggered, and a method is invoked within the application. In this simplified scenario, the application writes some output to a log. It should be evident that this type of middleware is typical of pervasive computing environments, and could underpin for example smart environments where functionalities are discovered only when a user moves into a certain proximity of a device or a feature.

5.3.1 Scenario Modelling requirements

From the PiCSE perspective, there are three "software" elements that require modelling. The two applications, `CommandSensor` and `TestDevice` must be instantiated individually, and are scheduled to execute at a parameterised time. These two applications both run on instances

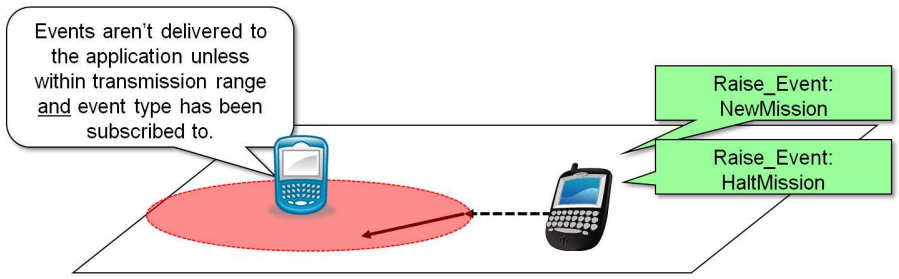

Figure 5.2: The CommandSensor application broadcasts events to his subscribers.

of the STEAM middleware, the third software abstraction, which must interact with the network simulator to model the wireless delivery of messages. As shown in figure 5.2 , the `CommandSensor` application raises events periodically, and broadcasts them in the wireless domain; the `TestDevice` application receives these events on its network interface whereby they are received by a listener and an appropriate handler is invoked.

There are two "hardware" modelling aspects of this scenario that map directly to the key abstractions within the PiCSE framework. The nodes are physical objects that have a location within the environment. As shown in figure 5.1, one node is mobile and moves according to the random waypoint movement pattern. The second "hardware" abstraction is the environment itself. In this scenario, the environment utilises PiCSE's default environment implementation, as no complex environment features are required. In effect, the environment is a free space in which the nodes can move freely.

In addition to the highlighted hardware and software components, an effective model of this scenario must address two high level features. These features include mobility and the modelling of the middleware functionality. Mobility is a key characteristic of STEAM applications. The scenario can only demonstrate the effectiveness of the STEAM middleware by realistically modelling the mobile components of the scenario. The STEAM middleware implements a range of functionality at different levels within its libraries. The scenario must retain the key parts of this functionality for the scenario to be a realistic validation of the STEAM middleware.

Function name	Functionality provided
`init_rte_publish()`	Initialises the Real Time Event publish component
`init_rte_subscribe()`	Initialises the Real Time Event subscribe component
`(un)announce()`	Broadcasts an event type announcement or denouncement
`send_event()`	Raises an event on the event channel
`(un)subscribe()`	Subscribes/Unsubscribes the middleware instance to a particular event type
`register_update_location_cb()`	Set a callback method where the current location can be obtained from
`various alloc_x() functions`	Various supporting functions for memory management etc.

Table 5.2: The STEAM API

5.3.2 Building the Scenario Model

Modelling the Scenario's Software Components

Modelling the STEAM Middleware

The original STEAM middleware is implemented as a library written in C++. The library comprises of two main components. An RTE library provides the core functionality for subscribing to events, announcing events and sending events, whilst the underlying second part of the library, Dummy-SEAR, provides functionality to both reserve and release channels for communication and to send serialised messages on those channels. The STEAM API, implemented in the C programming language, is provided allowing applications to be built upon STEAM and provides the functions outlined in table 5.2. The Dummy-SEAR aspect of the middleware interfaces with the operating system and network stack invoking the system calls listed in table 5.3.

In order to successfully emulate the STEAM middleware, its application stack, shown in figure 5.3 has to be re-written in terms of PiCSE components so that any of the API's upon which STEAM is built must be provided and replaced by the corresponding functionality provided by the PiCSE core. In the case of this particular communication middleware, this is primarily the networking functionality. There is some variability in how this might be achieved as there is a number of points within STEAM where an integration with PiCSE may

Function name	Source

`socket()`
`setsockopt()`
`bind()`
`sendto()`
`Recvfrom()`
	from sys/socket.h

`htons()`	from machine/endian.h
`inet_pton()`	from arpa/inet.h

`pthread_create()`
`pthread_detach()`
`pthread_exit()`
	from pthread.h

Table 5.3: System calls made by STEAM to the underlying operating system.

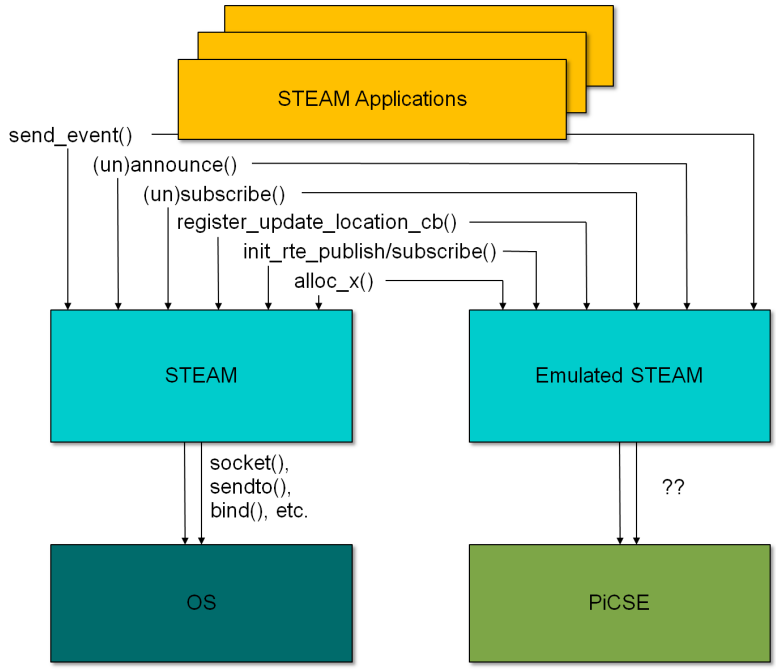

Figure 5.3: The appropriate point at which STEAM should be emulated must be determined by the user and is not constrained by PiCSE.

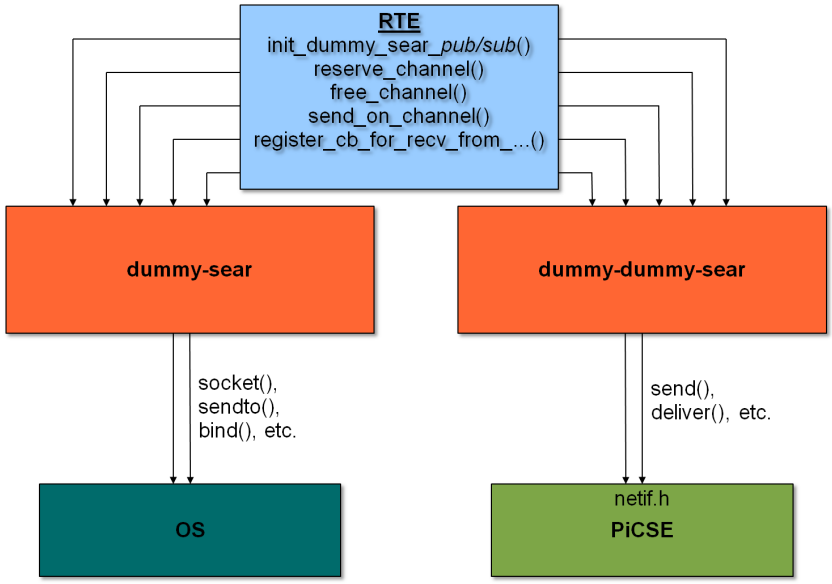

Figure 5.4: STEAM is comprised of an RTE and SUMMY_SEAR library.

be achieved. The point at which the middleware is emulated could be anywhere within the STEAM application stack, ranging from the API where the application interfaces directly with the middleware, to the API between the STEAM middleware and the underlying operating system. Choosing the appropriate point within that spectrum is a trade-off between several factors, which include:

- the maintenance of key functionality and logic provided within the middleware.
- the provision of any required simulated functionality within the PiCSE core.
- the factor which the simulation is ultimately trying to measure.

In the case of this evaluation, the aim is to show that the framework is flexible enough to emulate the middleware behaviour with as much fidelity as possible, and this is achieved by maintaining as much of the middleware's logic and functionality as possible. In the case of the STEAM middleware, shown in figure 5.4 , it is actually comprised of two component

DUMMY-SEAR function name
`init_dummy_sear_publish()`
`init_dummy_sear_subscribe()`
`reserve_channel()`
`reserve_channel_for_subscribe()`
`free_channel()`
`send_on_channel()`
`register_cb_for_recv_from_channel()`

Table 5.4: The DUMMY-SEAR API

parts: the RTE library and the DUMMY-SEAR library, where the RTE library is responsible for the event management, and the DUMMY-SEAR component is responsible for the timely serialisation, marshalling and transmission of messages containing events raised by STEAM applications. Since the event-based logic was self-contained within the RTE component, it is appropriate to leave that untouched. In doing so, any applications within the scenario that are built upon the STEAM middleware can run without modification, an important criteria for usability. The DUMMY-SEAR component primarily handles the network communication, and it was within this component where it was deemed appropriate to map functionality to the PiCSE core. DUMMY_SEAR provides the API, shown in table 5.4 which is accessed by RTE. The functionality abstracted by these API's must be modified to interface with PiCSE instead of the originally targeted operating system.

The two instances of the STEAM middleware are implemented as instances of the `SteamExecutionEnvironment` class, shown in figure 5.5 , which is derived from the `ExecutionEnvironment` abstraction. The `ExecutionEnvironment` abstraction, which itself implements the `Net_Interface` abstraction, transparently registers itself with the `Network_Simulator` as a wireless sender and receiver. Therefore, two basic methods have to be implemented. `Net_Interface::send(void*)` and `Net_Interface::deliver(void* msg)`. These methods interact directly with the network simulator in the PiCSE_Core, and it is via these methods that the PiCSE framework, both sends and delivers STEAM events between applications built upon the STEAM middleware.

By way of example, figure 5.6 shows the methods invoked when an application sends an event via the STEAM middleware. Part a) traces the method calls through the RTE,

```
#ifndef STEAM_EXEC_ENV
#define STEAM_EXEC_ENV
#include "execEnv.h"
class SteamExecutionEnvironment:public ExecutionEnvironment{
 public:
  SteamExecutionEnvironment(Location l);
  ~SteamExecutionEnvironment();
  // RTE - Functions called from RTE_Proximity library
  int init_rte_publish_local();
  int init_rte_subscribe_local();
  channel_id announce_local(subject sub, proximity* prox,
            latency lat, period per,adaptation adapt);
  void unannounce_local(channel_id ch);
  int send_event_local(int channel_id, event* event);
  subscription_id subscribe_local(subject sub, receive_event cb);
  void unsubscribe_local(subscription_id ch);
  void register_update_location_cb_local(update_location_cb cb);
  void make_callbacks_for_local(steam_event* st_ev);
  void free_steam_event_local(steam_event* ev);
 // DUMMY-NETIF - Functions called from DUMMY^2-SEAR functions
  void send(void*);
  void deliver(void*);
 // DUMMY^2-SEAR - Variables needed for DUMMY^2-SEAR functions
  int send_socket;
  int recv_socket;
};
#endif
```

Figure 5.5: Extracts from the emulated STEAM library, as defined in steamExecEnv.h. The full definition of this program may be found in the Appendix.

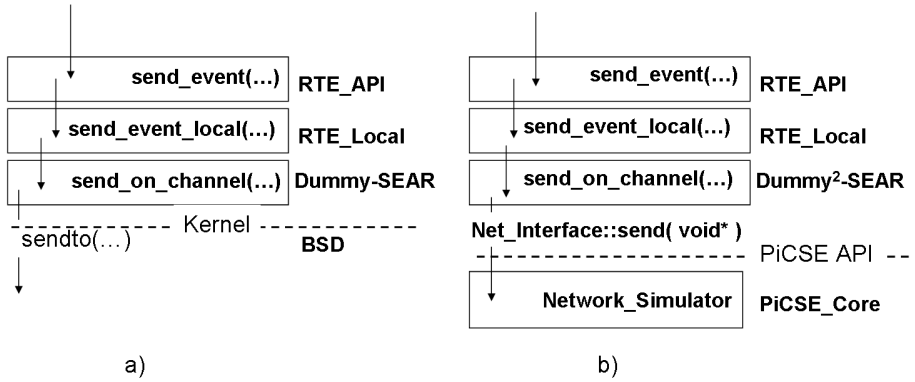

Figure 5.6: Original STEAM library and emulated STEAM library

and DUMMY-SEAR libraries respectively before the event is passed to the underlying operating system. In part b), the DUMMY_SEAR library has been replaced with the DUMMY-SEAR library which was implemented in the SteamExecutionEnvironment class. The emulated send_on_channel method sends the event to the network simulator via the Net_Interface::send method. Beyond that method, the network simulation component of the PiCSE framework calculates the potential receivers of that message based on proximity and delivers that message to those receivers at a scheduled point in the future. The deliver of messages at a time in the future is enabled to allow developers to simulate packet delays, but this is not a feature of this particular scenario.

Figure 5.7 shows the modifications required to each of these functions in order to redirect these functions from making calls to the underlying operating system to the PiCSE core. The left column shows the original functionality provided within the DUMMY-SEAR library, and the right hand column shows the functionality provided by the emulated library, named DUMMY^2-SEAR library, and the calls that are made to the PiCSE libraries in order to achieve that emulation.

Modelling the Scenario's Applications

There are two applications, `CommandSensor` and `TestDevice`, in this scenario. Both are supported by their respective `ApplicationWrapper` instantiations, `Command_Wrap` and

DUMMY-SEAR MW component	Emulated DUMMY²-SEAR MW Component
init_dummy_sear_publish() - Calls *get_send_socket()*, which: ● Creates socket (*send_socket*) that can broadcast & reuse local addresses ● Creates a broadcast address.	**init_dummy_sear_publish()** - Sets *send_socket* to 0. (dummy-sear uses *send_socket* for all channel Ids.) - Creates a new network interface *send_nif* for sending events.
reserve_channel() - Creates an event channel using the given parameters. - Sets channel ID to *send_socket*. - If RANDOM_ADAPT is defined, then a new thread is created running function *random_adapt()*, which: ● Loops infinitely, changing the actual proximity of the channel every few seconds.	**reserve_channel()** Should be pretty much the same as the dummy-sear version. (Perhaps removing *random_adapt()*.)
init_dummy_sear_subscribe() - Calls *get_recv_socket()*, which: ● Creates socket (*recv_socket*) that can broadcast & reuse local addresses ● Creates a broadcast address. ● Binds broadcast address to the socket. - Creates a new thread running function *select_for_recv()*, which: ● Loops infinitely, checking if the *recv_socket* is ready to read from. ● If it is, calls *recvfrom()* on the socket & then sends the buffer to a new thread running *recv_event()*, which: ■ Unmarshals the buffer into a STEAM event. ■ Calls *make_callbacks_for()* from RTE. ■ Frees the event and buffer.	**init_dummy_sear_subscribe()** - Sets *recv_socket* to 0. (dummy-sear only uses *recv_socket* in *select_for_recv()*.) - Creates a new network interface *recv_nif* for receiving events. **dummy_netif::deliver()** *Gets called by Net_interface whenever an event is sent by any network interface but recv_nif.* - Calls *make_callbacks_for()* from RTE. - Frees memory used by the event.
reserve_channel_for_subscribe() - Creates an event channel using the parameter for the subject, *send_socket* for the channel ID and 0 for the rest of the attributes.	**reserve_channel_for_subscribe()** Should be pretty much the same as the dummy-sear version.
free_channel() - Frees the memory used by the subject and proximities of the given channel. - Frees the memory used by the given channel.	**free_channel()** Should be pretty much the same as the dummy-sear version.
send_on_channel() - Marshals the given steam event into a string. - Calls *sendto()* on the socket, with the marshalled string and the channel ID of the given channel (which is always *send_socket*). - Frees the memory used by the marshalled event.	**send_on_channel()** - Calls *dummy_netif::send()* on *send_nif* and sends a copy of the given event casted to a void pointer. **dummy_netif::send()** - Calls *Net_interface::send()* to send the given event.
register_cb_for_recv_from_channel() - Does nothing.	**register_cb_for_recv_from_channel()** - As original, does nothing.

Figure 5.7: Comparing original behaviour of the Dummy-SEAR and the emulated DUMMY²-SEAR components.

```
CommandSensor::CommandSensor(){
if(init_rte_publish() == -1){
  printf("Init-rte-publish failed.\n");
  exit(1);
}
my_proximity = alloc_proximity();
my_proximity->proximity_hull.type = CIRCLE;
// non-rt latency
lat.min = 0; lat.max = 0;
//period is in milliseconds
per = 1000;
strcpy(newMissionSubject, "NewMission");
channel_id newMissionChannel;
newMissionChannel = announce(newMissionSubject, my_proximity, lat, per, commandSensor_ada
newMission_event = alloc_event();
newMission_event->sub = newMissionSubject;
int xv = 0 ;
while (xv < 10){
  sleep(7);
  if(send_event(newMissionChannel, newMission_event) == -1){
    printf("Error in send_event\n");
    exit(1);
  }
  xv++;
}
}
```

Figure 5.8: Extracts from the CommandSensor program, which is defined in CommandSensor.cpp

Test_Wrap and these instantiations initialise the CommandSensor, and TestDevice programs respectively. The CommandSensor program is a simple program that exercises the STEAM event production API. In this scenario, this program initialises itself by calling the appropriate RTE library methods, and then sends two separate events on the created STEAM event channel. As can be seen in figure 5.8 , these messages are raised periodically every seven seconds and are sent ten times in total. The TestDevice program, an abbreviated version of which is shown in figure 5.9 , is a simple STEAM event consumer, that listens for "NewMission" events by registering a callback function shown in figure 5.10 . It is the STEAM middleware that in-

```
TestDevice::TestDevice(){
  if(init_rte_subscribe() == -1){
    printf("Init-rte-subscribe failed.\n");
    exit(1);
  }
  if(init_rte_publish() == -1){
    printf("Init-rte-publish failed.\n");
    exit(1);
  }
  testDevice_cur_loc.lat = 0;
  testDevice_cur_loc.lon = 0;
  subscribe("NewMission", testDevice_recvev);
  printf("End\n");
}
```

Figure 5.9: Extracts from the TestDevice program, which is defined in TestDevice.cpp

```
void testDevice_recvev(event* ev){
  printf("THIS IS THE TEST DEVICE RECEIVING AN EVENT\n");
  printf("¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬\n");
  printf("Received event of subject \"%s\", with
  content:\n[%s]\n", ev->sub, ev->cont);
  printf("¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬\n");
}
```

Figure 5.10: The TestDevice event callback function, which is defined in TestDevice.cpp

```
SteamExecutionEnvironment* testDeviceSee =
new SteamExecutionEnvironment(Location(0,0));
```

Figure 5.11: Modelling of a static object can be achieve by passing a location parameter during the instantiation of an ExecutionEnvironment object.

vokes that callback, which is triggered by the delivery of a "NewMission" Event from the PiCSE network simulator to the middleware instance through its `Net_Interface::deliver(void*)` method. The full definition of both the TestDevice and CommandSensor programs can be found in the Appendix.

Modelling the Scenario's Hardware Components

Modelling Static Objects

There are two physical components in this scenario: the static physical device upon the TestDevice program is running, and the mobile physical device on which the CommandSensor program is running. The first physical device is specified in the scenario to be a static object, initiated at a fixed location, which remains at that location throughout the duration of the experiment. This can be done in PiCSE using two methods. The first method is to use a `Mobility_Object` class, which may be parameterised at instantiation with the STATIC enumerator. Alternatively the ExecutionEnvironment can be parameterised with a location at the time of instantiation. In the implementation of the STEAM scenario, the second approach was utilised as only a single line of code is required to initialise this, and this can be additionally achieved by passing a location parameter during the creation of the scenario ExecutionEnvironment as seen in figure 5.11[1] .

Modelling Mobile Objects

An instance of a `Mobility_Object` is instantiated and parameterised with the RANDOM_WAYPOINT parameter to model the mobility of the device hosting the Command-

[1] Variable names may have been renamed in code extracts in order to facilitate readability and understanding of the scenario.

```
Mobility_Object* commandSensorMobObj =
new Mobility_Object( Location(0,0), RANDOM_WAYPOINT ) ;
```

Figure 5.12: Physical objects can be parameterised with a mobility pattern which defines their movement during the scenario experiment.

Sensor program. When a `Mobility_Object` is initialised with this parameter, shown in figure 5.12 , its path is determined using the random waypoint algorithm. This is a well known and frequently used algorithm in network simulation experiments in which an object's mobility pattern is guided by a series of waypoints, the next of which is determined upon arriving at the previous waypoint. MOBILITY events are transparently scheduled by the PiCSE simulator to move the `Mobility_Object` towards its next waypoint. By default, these two object instances are added to the default environment, `default_entLayer`, which is managed by the `Domain_Manager` component.

Interlinking of Simulated Hardware and Software Components

The final step in the modelling of the STEAM scenario is to interlink the individual hardware and software components that have been described and instantiated in sections 5.3.2 and 5.3.2 of this chapter. So far, these components have been created in isolation: that is to say, that the components such as an emulated middleware instance and a modelled physical device that has a location are created but are not yet interlinked in a meaningful way. However, applications in PiCSE, handled by their `ApplicationWrapper` instantiations have to be associated with their respective `ExecutionEnvironment` instantiations before they can be meaningfully executed.

In addition to emulating the middleware, the `SteamExecutionEnvironment` instantiation has a physical location within the simulated world. As this scenario contains mobile nodes, the location of the instantiation, and the implicit location of all applications running on top of that instantiation, have to reflect the location of the mobile nodes. Therefore an instantiation must be associated with a `Mobility_Object` instance that models its physical node. If that `Mobility_Object` moves within the simulated environment, the location of the `ExecutionEnvironment` instantiation should be correspondingly updated automatically.

This is achieved within PiCSE by the `Location_Manager` component of the PiCSE core. The `Network_Simulator` is also informed of the updated position of the `ExecutionEnvironment` instance.

Figures 5.13 and 5.14 show code fragments from the initialisation of the scenario experiment in which the instances, that been been described in sections 5.3.2 and 5.3.2, are now implemented and interlinked.

As the code fragment in figure 5.13 shows, the emulated middleware is instantiated in a

```
1. SteamExecutionEnvironment* testDeviceSee =
   new SteamExecutionEnvironment(Location(0,0));
2. thirdprog* testDeviceWrapper = new thirdprog();
3. testDeviceSee->addEmulatedApplication(testDeviceWrapper);
```

Figure 5.13: Code fragment demonstrating the interlinking of the scenario's TestDevice application and emulated STEAM middleware instance.

single line, line 1. The application TestDevice is instantiated through its application wrapper, and is also instantiated in a single line, line 2. The interlinking of these components is achieved in a single line, line 3, whereby the program, mediated by its application wrapper is 'added' to the middleware through the addEmulatedApplication method. This method associates the program and any of its underlying middleware calls to a single emulated STEAM middleware instance, in this case, named testDeviceSee. This "hook" allows application events that are raised as part of its execution to be pushed down into the middleware, and conversely allows events to be delivered from the middleware to the application.

Figure 5.14 shows how the interlinking of software and hardware components is achieved for a mobile application, the CommandSensor program and middleware instance in this scenario. The application and the middleware instance are created similarly to the TestDevice instantiation in lines 1,3 and 4. Two further lines of code are required to capture the interlinking of these components. A Mobility_Object named commandSensorMobObj, is instantiated in line 2 as described in section 5.3.2, and the program's execution environment, the emulated middleware instance commandSensorSee, is now linked to that Mobility_Object instance using the addObject() method.

```
1. SteamExecutionEnvironment* commandSensorSee =
   new SteamExecutionEnvironment(Location(0,0));
2. Mobility_Object* commandSensorMobObj =
   new Mobility_Object( Location(0,0), RANDOM_WAYPOINT ) ;
3. prog* commandSensorWrapper = new prog();
4. commandSensorSee->addEmulatedApplication(commandSensorWrapper);
5. commandSensorMobObj->addObject(commandSensorSee);
```

Figure 5.14: Code fragment demonstrating the interlinking of the scenario's CommandSensor application, emulated STEAM middleware instance and the modelled mobile devices

This is achieved using PiCSE events. The `Mobility_Object` periodically schedules MOBILITY events. Each event occurrence invokes a method that moves the `Mobility_Object` towards its next waypoint. Upon reaching the waypoint, the node chooses a waiting period, and its next waypoint, and schedules another MOBILITY event at the end of the waiting period. As the `Mobility_Object` moves, as determined by the parameterised random waypoint algorithm, any objects co-located with that object, in this case the emulated STEAM middleware instance and the CommandSensor application running on that middleware, also move according to the same random waypoint algorithm. Simulated object mobility is thus achieved when the experiment is executed.

5.3.3 Executing the Scenario Simulation

Initialising the Experimental Scenario

As in all of the scenarios described in this chapter, an instantiation of the *PiCSE_Core* class category must exist and enable the simulation of the scenario. The primary role of the *PiCSE_Core* in this instantiation, is to manage the models of the nodes, executing scheduled events when required, and to manage the execution of the two applications, the CommandSensor and TestDevice programs. The *PiCSE_Core*'s network manager also models the wireless network communications between the applications which is mediated by the STEAM middleware.

The *PiCSE_Core* is the first set of components to be instantiated, and the follow-

```
int main( int argc, char **argv ){
  ..
  // Create a simulation engine
  new Future(LINKED); // event list, clock, etc
  Simulator::Instance();
  // Instantiate the network
  Network::Instance();
  // Create a world
  World::Instance();
  ..
}
```

Figure 5.15: Code fragment showing the initialisation of the PiCSE_Core aspects of the STEAM scenario environment.

ing components are created by the code snippets shown in figure 5.15 . The key components are the `Domain_Manager`, the `Network_Simulator` and the `Event_List_Manager`. The `Domain_Manager` initialises a world model, instantiating an empty `EntityLayer`, default_entLayer, by which all `Object` instantiations can be referenced. The `Network_Simulator` creates an `EntityLayer` that maintains the position of all `Net_Interface` instantiations, that is instantiations that can send and receive messages. Finally, the `Event_List_Manager` instantiates two event lists, the `Scheduled_Event` list and `Scheduled_Application` list. These lists are also empty by default.

In addition to the PiCSE core components, a range of scenario specific parameters have to be defined. Figure 5.16 shows an excerpt from the main.cpp, which defines the experiment's initialisation main method. The experiment's duration is specified in milliseconds; the physical size of the experimental space is defined as a Cartesian space of one square kilometre, and the experiment is defined to run over a single iteration.

The scheduling of the execution of the simulated applications at a specified period after the beginning of the simulation, finally completes the definition and initialisation of the STEAM scenario and the scenario experiment is now ready to be executed. In this scenario, the two applications are scheduled to start at staggered intervals in order to highlight behaviour when one of the applications is not running.

```
int main( int argc, char **argv )
{
  int simDuration = 1000*60*2.5;
  int iterations = 1;
  int worldXSize = 1000, worldYSize = 1000;
  // Set the size of that world
  World::setXSize(worldXSize);  World::setYSize(worldYSize);
  // Set simulation duration
  Simulator::setDuration(simDuration);
  // Schedule the execution of emulated applications
  testDeviceWrapper->scheduleExecution(1970);
  commandSensorWrapper->scheduleExecution(1980);
```

Figure 5.16: Code fragment showing the initialisation of the STEAM scenario environment.

Execution of the Simulation

The experiment was performed on an off-the-shelf Dell Latitude C400 laptop, equipped with a minimal 512 MB of RAM and running on Ubuntu 6.10, Edgy Eft, an open source operating system built around the Linux kernel.

The experiment was executed within a terminal and figure 5.17 shows an excerpt of the logged output of the experiment. A more complete excerpt of the logged output is available in the appendix. The excerpt in figure 5.17 shows the two programs simulated exchange and delivery of events via the STEAM middleware after the initialisation of the experiment, and the initialisation of the programs, and the figure is annotated with a series of pop-up notes, which explain the outputs of the experiment.

The `CommandSensor` program, logically executing on the `Mobility_Object` raises two events, "NewMission" and "HaltMission" every seven seconds. The TestDevice program, executing in a static environment, consumes "NewMission" events only. A number of events of interest should be noted in figures 5.17 and 5.3.3 .

In notes 4 and 5 of figure 5.17, the TestDevice program receives two messages: a "HaltMission" announcement and a "NewMission" event. The "HaltMission" is received and processed by the middleware. The middleware checks its subscription lists for subscribed applications, but none are found for events of this type so no callbacks are made to the TestDevice appli-

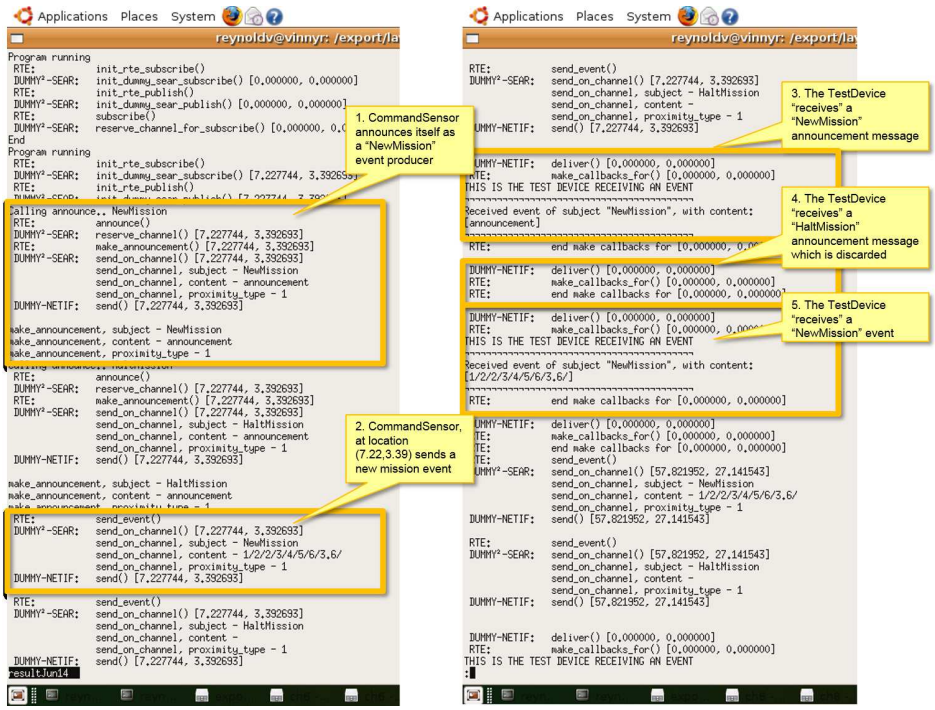

Figure 5.17: A screenshot showing the logged output from the STEAM emulation evaluation.

cation. In contrast, the "NewMission" event is an event type for which a callback has been registered, so when the middleware checks its subscription lists for subscribed applications, it finds the registered callback method within the TestDevice application, which in this scenario prints the output seen in note 5.

Notes 6 and 7 demonstrate another feature of the STEAM middleware. In note 6, the CommandSensor raises another "NewMission" event and transmits it from the location (108, 50). Note 7 shows what happens at the TestDevice's STEAM middleware instance upon receiving that event. Although the program has subscribed to these event types, no callback is made and the event is effectively discarded as the CommandSensor has defined that the message is only to be delivered if the receiver is within a certain proximity. In this particular case, the CommandSensor has defined the proximity to be 100 metres and the distance from the sender to the receiver is approximately 119 metres. The proximity is defined within the CommandSensor as:

```
my_proximity->proximity_circle.radius = 100;
```

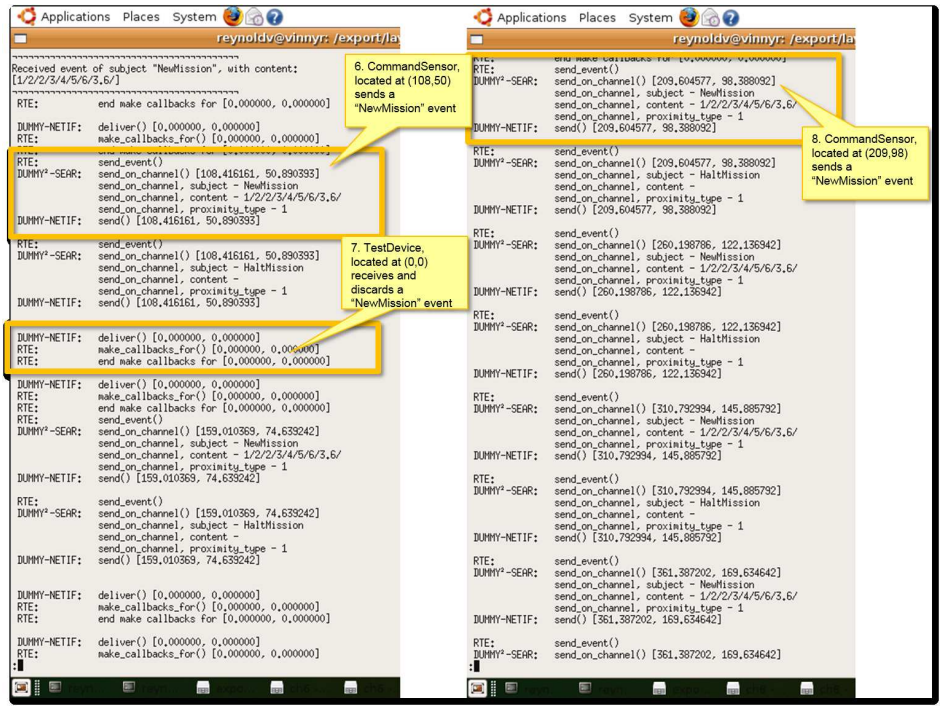

Figure 5.18: A secondary screenshot showing the logged output from the STEAM emulation evaluation.

Finally, note 8 of figure demonstrates the range transmission aspect of the network simulator component of the PiCSE simulator. The CommandSensor transmits a "NewMission" event at the location (209,98), but the event is not received by the TestDevice middleware as the sender in this case is approximately 231 metres from the only potential receiver. The default transmission range within the simulator is 200 metres, and therefore the message is not delivered.

5.3.4 Scenario Analysis

Several of PiCSE's abstractions have been exercised in this scenario. In the modelling of the location of the `ExecutionEnvironment` instantiation, a physical dependency was created using the underlying `Object` abstraction. `Mobility_Object` components, and `ExecutionEnvironment` instantiations are both derivations of the `Object` abstraction. The modelling of the co-location of a middleware instance and a physical mobile node in this scenario, demonstrates the abstraction's support for modelling physical dependencies. This dependency is independent of the actual `Object` instantiations, thus validating the abstract approach and the capturing of this dependency contributes to accurately fulfilling the modelling of one of the scenario's key features, that of mobility.

The `Net_Interface` abstraction is implemented by default for all `ExecutionEnvironment` instantiations. The API allows a range of message types to be passed through the simulator, irrespective of the application or middleware that is sending and receiving those messages. In this scenario, the `SteamExecutionEnvironment` instantiation uses the abstractions interface to send and receive STEAM events. The outputs of the scenario simulation demonstrate the rudimentary functionality of a network simulator, and show how a event-based messaging middleware may be integrated into that network simulation component.

In the case of the STEAM middleware, the event subscription and publication APIs, in addition to the supporting methods required were deemed to be central to the accurate modelling of the STEAM scenario. The Dummy-SEAR aspect of the library was not central to the application scenario, so the emulation point was chosen appropriately. The location of the emulation point at this point preserves the key middleware functionality, and this is the

second feature of the STEAM scenario that must be supported.

Overall, from a domain perspective, the simulation of this scenario was a success. The STEAM scenario's two most important features, the capturing of mobility and its effects on the behaviour of the applications, and the retention of the key functionality of the middleware have both been fulfilled. In both cases, this has been achieved using features of the PiCSE framework. Physical dependencies have been demonstrated to work as a methodology for modelling co-location, enabling the simulation of realistic mobility-based scenarios. Similarly, the ExecutionEnvironment abstraction has been demonstrated that it can be used to effectively emulate middleware, at least of the event-based communication variety.

Insights

The primary effort in this scenario was in the implementation of the emulation of the STEAM middleware, and this is where the most relevant and important observations are to be made.

The most important learning is around how much effort is required to provide the emulated middleware. Conceivably, almost any of the layers, shown in part a) of figure 5.19 could have

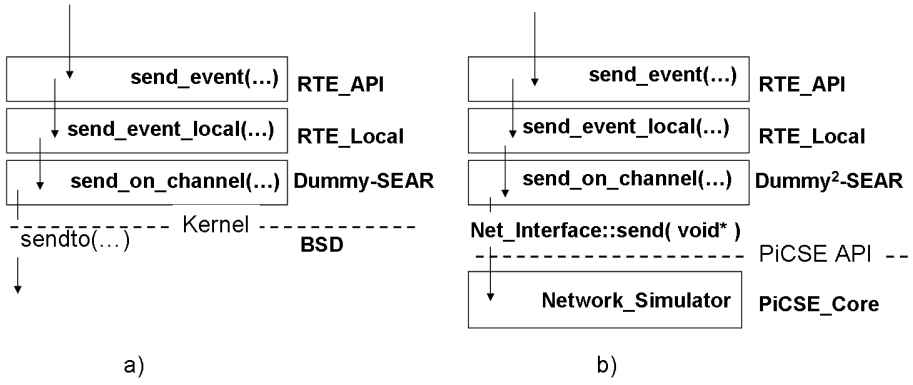

Figure 5.19: Original STEAM library and emulated STEAM library

been chosen as the point at which to emulate. Consider the selection of the emulation point by examining the API from top to bottom. A trade-off occurs as the emulation point descends into the lower levels of the middleware implementation. As the emulation point lowers, a greater amount of the existing middleware functionality is maintained, however there are also

more likely to be operating system calls invoked above the emulation point. These system calls must additionally be emulated if they remain in the middleware implementation.

In the case of this experiment, the emulation point chosen removed the need for pthreads to be implemented as part of the emulation. In contrast, one system call, `extern unsigned int sleep (unsigned int __seconds)` remained above the emulation point, and this system call was emulated in order to preserve the functionality of the middleware. The choosing of the emulation point for any emulated applications or middlewares should be considered as part of a domain specific cost-benefit analysis that should be addressed by any user of the PiCSE framework.

Some restrictions were noted in the overall approach taken to incorporating applications developed by third parties. At present, only programs which are developed in c++ are incorporated. Bindings could theoretically be made available in alternative languages such as java, and python, but this has not yet been tested. This would be a significant amount of work involved in such an undertaking, and a re-architecting of the PiCSE core to represent a more distributed simulation environment would be required. In such an architecture, emulated programs would likely be executing within a lightweight local PiCSE instantiation and the management of these instantiations could be coordinated and synchronised through a centralised instantiation.

As PiCSE does not provide a sand-box, or a virtual environment based approach, some changes have to be made to applications before they can be implemented. Within the scope of applications that are developed in c++, several changes had to be made to the applications before they could be embedded into a PiCSE simulation. In c++, the program is invoked through a single main() method. As an instantiation of the PiCSE framework already contains the only available main() method, the main() method of any emulated applications must be renamed.

In a similar vein, some c++ keywords that are used within a program may have to be renamed. For example, static variables should be renamed, for example pre-pended with a namespace such as the applicationWrapper instance, in order to ensure there are no conflicts with other emulated programs.

	Definition (.h)	Implementation (.cpp)
steamExecEnv	67	477
testDevice	34	91
commandSensor	36	199
main		61

Table 5.5: Lines of code required to implement the STEAM emulation scenario

In analysing the effort required to implement this scenario, the codebase is considered in three parts.

1. The lines of code required to emulate the core aspects of the middleware.

2. The lines of code required to implement the two applications, TestDevice and CommandSensor, that exercise that middleware.

3. The lines of code that are required to construct and initialise the scenario.

The lines of code methodology is chosen over determining the time taken to implement the scenario as that methodology can be subjective and skewed by factors such as the knowledge of a user of a system, their level of domain expertise, and their overall skill as a developer. The number of lines of code is in itself a subjective approach for measuring effort, but arguably less so than the time taken. Table 5.5 shows the lines of code required to implement the scenario as considered when broken into the three parts. It can be seen that the greatest effort, at least as measured by the total lines of code written, is in the implementation of the emulated middleware. Figure 5.20 examines the breakdown of the salient aspects of that implementation. By comparing the lines of code in the original implementation and in the emulated implementation, it can be seen where the focus of attempting to accurately emulate the middleware was placed. The implementation of methods such as get_recv_socket() and get_send_socket() were forfeited, whilst free_channel() and send_on_channel() were reimplemented. These reflect the point at which the middleware was emulated at. Simulating network behaviour is forfeited whilst the modelling of the channel-based communication behaviour of the middleware is maintained.

Function Name	Dummy-SEAR	DUMMY²-SEAR MW Component
get_recv_socket() *[init_dummy_sear_subscribe()]*	***19***	*Eliminated*
get_send_socket() *[init_dummy_sear_publish()]*	*15*	*Eliminated*
recv_event() *[select_for_recv()]*	*8*	*Eliminated*
init_dummy_sear_subscribe()	*7*	*5*
init_dummy_sear_publish()	*3*	*5*
select_for_recv() *[dummy_netif::deliver()]*	*21*	*Eliminated*
reserve_channel()	*18*	*13*
random_adapt()	*13*	*Eliminated*
reserve_channel_for_subscribe()	*10*	*10 [No Changes]*
free_channel()	*4*	*4 [No Changes]*
send_on_channel()	*10*	*15*
register_cb_for_recv_from_channel()	*No code*	*No code*
dummy_netif::deliver()	*n/a*	*2*
Total	***128***	***54***

Figure 5.20: Table showing the number of modified lines of code with the emulated DUMMY-SEAR library.

Overall, the number of lines of code required to implement those changes is demonstrated in this scenario at least to be relatively small and in most cases in line with the original implementation of the relevant aspects of the middleware. This is broadly positive as it demonstrates that in these cases, a like-by-like substitution can be made when replacing the original line of code with a line of code that invokes the simulated functionality within PiCSE. Furthermore, the effort required to implement the emulated STEAM middleware is a single up-front effort that needs to be theoretically implemented once and then emulated middleware can then be used in other scenarios without modification.

The actual behaviour within the scenario is defined through a combination of the TestDevice and CommandSensor applications, implemented in 125 and 235 lines of code respectively. The experiment was initialised and the world and models created in the main.cpp file and are implemented in 61 lines of code. The number of lines of code to define the experimental behaviour and to initialise and construct the scenario is relatively small in this particular scenario as its focus was to demonstrate the feasibility of the middleware emulation aspect of the PiCSE core. In this scenario, existing components were used to exercise the mobility models for example, and no custom hardware components were defined. As such, the small number of lines of code is more a reflection of the non-complexity of the scenario rather than a statement suggesting that any scenario can be constructed in a similarly small number of lines of code.

This scenario has not been replicated in the real world setting so no comparison can be made between the simulated behaviour and observed real behaviour. However some observations and insights can be drawn based on the running of the simulation alone.

Overall the emulated applications and the STEAM middleware instances have behaved as expected. Within the middleware, a full range of its functionality was observed, from within the middleware and up to the applications that were running on that middleware. Within the physical environment that was constructed within this scenario, the models for the physical aspects of the simulator have moved and interacted as expected. Finally, the network simulation component has been seen to deliver messages as expected whilst respecting expected scenario characteristics such as mobility traits and transmission ranges.

However, there are several features that are missing from this experiment that an observer would expect to see had the experiment been conducted in a real world setting. For example, by removing the part of the library that was built upon the pthread library, some non-deterministic aspects of the middleware's behaviour have been removed. In a more complex scenario with greater interaction of its components, increased demand for resources within a middleware instance would have affected the execution and behaviour of that middleware. In a real world setting, one would also find that all packets sent by the communication middleware across the network would not have been delivered due to interference or other issues not captured by our PiCSE's basic network simulation component. For example, the transmission range of a wireless network is not an absolute fixed number and is influenced by a number of factors such as the mobility of the objects and occlusion created by buildings and users carrying devices.

Finally scenario specific behaviour that occurs in the real world such as the positioning of external components such as buildings, and mechanical aspects of components that are represented in this scenario as mobility objects are also missing. Similarly, the scenario presumes that the applications are the sole running applications on their host device, and therefore the scenario does not take into account some features such as demand from other applications for computing resources of that host device.

Although the STEAM emulation scenario as described does not account for these "real-world" concerns, the scenario could have been extended to take these aspects into account. The scheduling component of the PiCSE core has a facility for delaying the execution of an application, allowing for some rudimentary simulation of the delayed or even failed execution of an application. The network simulation component can drop packets probabilistically, and can be integrated into a third party network simulator to take advantage of existing proven network simulation models. Finally, all physical artefacts within this scenario can be extended to take into account aspects of physical components such as failure and human behaviour. Indeed, the next two scenarios that are presented examine some of these aspects.

Metrics	STEAM Emulation scenario
The number of diverse instances of all hardware components.[1]	1 object type, 1 environment type
The number of instances of emulated components, both applications and middlewares.[1]	2 applications, 1 middleware instance
The number of instances of component inter-relationships, which is defined as an interaction between two separately created instances of PiCSE abstractions.[2]	1 object-environment relationship, 1 application-middleware relationship, 1 application-object relationship
The number of instances of inherited reusable abstractions. Measure the proportion of the simulation that is comprised of re-usable domain specific elements.	1 middleware instance 53.1% of 898 LOC
The proportion of the simulation that is provided by the PiCSE frameworks core classes.[3]	79.7% of 4429 LOC
Demonstration of network functionality in application scenarios.	TRUE
Demonstration of evidence of evaluation support functionality.	FALSE

Table 5.6: Requirements analysis for the STEAM emulation scenario

Evaluating the PiCSE requirements

The outcomes of this scenario and its subsequent analysis are summarised in table 5.6 . The requirements are measured with respect to their associated metrics which were identified in section 5.2 of this chapter. This scenario provides a demonstration of almost all of the basic functionalities provided by the PiCSE framework. Instances of almost all of the main software and hardware abstractions were created and some heterogeneous interactions between some of these instantiations were also demonstrated. Note 3 of this requirements analysis highlights that the framework can provide up to 80% of the effort required in creating the basic components that this typical pervasive computing scenario requires.

A final over-arching analysis of the requirement satisfaction across all three scenarios is completed in the final section of this chapter.

Figure 5.21: Photograph of the hardware being emulated.

5.4 The Car Hardware Scenario

The car hardware scenario is the second scenario used to validate the PiCSE framework. The real-world equivalent of this scenario was first described in (Senart 06) and was presented as a potential application for the MoCoA framework, a software framework for supporting the development of context aware applications. This application scenario envisages a sentient robot, an autonomous mobile robot with onboard sensors, which is controlled by an application developed using the MoCoA framework, with the ability to read sensor data from those sensors and then reason and act upon that sensor data.

A prototype of this scenario was constructed using the hardware shown in figure 5.21 . Two sensors, a compass and a sonar sensor are attached to a PIC micro-controller board which is physically attached to a re-purposed remote-control car. These sensors periodically measure the distance to, and the relative orientation of the location of any detected obstacles. An application hosted on an IPAQ PDA device, running a slim-line Linux distribution, implements low-level functionality that reads the raw sensor data from the abstract Linux file-system.

The embedded nature of this scenario, albeit in the context of a mobile robot is typical of

pervasive computing scenarios. Sensor data samples real world phenomenon and is consumed and reasoned upon, until some actionable decision is reached and some changes are affected, sometimes with the application itself as is the case in context-aware computing, but often in the environment. This is more typically seen in ambient assisted living or smart space scenarios. In contrast to the STEAM scenario presented in section 5.3, this scenario was chosen as it reflected the sometimes embedded nature of pervasive computing applications. In these scenarios, applications are typically no longer written upon middleware frameworks but directly upon application programming interfaces provided by an operating system.

5.4.1 Scenario Modelling requirements

The high level set up of this scenario is shown in figure 5.22 . The modelling of the emulated

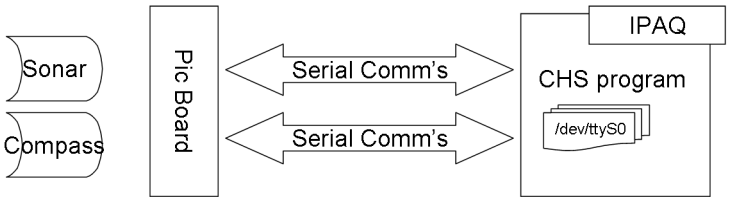

Figure 5.22: The hardware setup of the Car Hardware Scenario. The program running on the IPAQ, consumes sensor data via the PIC board.

interaction between the hardware and software components, which is mediated by an emulated Linux operating system is captured and is the primary focus of this scenario. By modelling this interaction, it will be demonstrated that PiCSE can emulate certain pervasive computing applications that are built directly upon API's of an operating system as well as emulating applications that are built upon pervasive computing middlewares.

Two features of this scenario must be modelled by the PiCSE instantiation in order for the evaluation scenario to be validated.

Emulation of system call based programming interfaces. This scenario should validate the framework's ability to emulate at the level of system calls. The scenario should

therefore include an application that is built upon a representative sample of these types of system calls.

Realistic modelling of the interaction between simulated and emulated hardware components. The realistic modelling of the interaction and behaviour between the sensors and the application is critical to this scenario. In order to be able to state that the PiCSE framework can emulate these hardware components, the application in this scenario should behave as if it was interacting with real sensor devices and do so without modification.

The scenario's modelling requirements are completed by imagining a third party viewer who wishes to observe and capture aspects of the scenario for offline post-experimental analysis. The scenario should demonstrate the functionality required to support such a viewer and record his events of interest.

5.4.2 Building the Scenario

There are three physical and two software modelling aspects to this scenario. The two sensors are implemented as instantiations of the `Sensor` abstraction. An instantiation of an `EntityLayer` abstraction models the location of obstacles that the sonar sensor can detect. A `Mobility_Object` is used to model the remote-controlled vehicle. The Linux operating system, on which the IPAQ application runs, is captured as an instantiation of the `ExecutionEnvironment` abstraction, while the IPAQ application itself, is incorporated using an instantiation of the `ApplicationWrapper` abstraction.

Modelling the scenario's software components

The IPAQ application

The program that is running on the IPAQ is implemented as a single class that includes an infinite while-loop which is an a suspended state until a sensor reading is created and the application is notified. The application then reads and validates the sensor reading and raises an event which could then be consumed by a separate navigation module. In the simulation of

```
27.   fd = open(DEVICE, O_RDWR | O_NOCTTY | O_NDELAY);
84.   rv = select(fd+1, &read_fds, NULL, NULL, NULL);
87.   res = read(fd, buffer, 1);
249.  close(fd);
```

Figure 5.23: System calls made by the IPAQ application

this scenario, the focus is on the realistic emulation and interaction between the sensors and the application so the navigation module and its logic is not included as part of this scenario.

Within PiCSE, this application is handled in the normal way. An instantiation of the `ApplicationWrapper` class, `Chs_Wrap`, initialises the IPAQ application by invoking the appropriate method of the application class, in this case, its constructor, CarHardwareSensor().

The application's while-loop is implemented as a single thread that is waiting on character events to occur on an open filestream. The thread is blocked while no new characters are available at that file-descriptor and is only awakened when a sensor pushs a character to that descriptor. The application makes references to four different system calls within its implementation and these are listed in figure 5.23 . These system calls are the bridge between the application and its sensors, via all used to interact with the underlying file-system. In order for the emulated version of this application to behave without modification, these system calls should be implemented and emulated by some aspect of the PiCSE framework. Fortunately, this is handled by a new instantiation of the `ExecutionEnvironment` class.

The Linux_ExecutionEnvironment Instantiation

An instantiation class of the `ExecutionEnvironment` abstraction, `LinuxExecutionEnvironment`, is implemented to emulate the Linux operating system. This instantiation class provides the emulated equivalent of the system calls that are made by the IPAQ application. The mapping between these system calls and their emulated equivalent is listed in table 5.7 . For example, the application invokes the system call

```
int open (__const char* __file, int __oflag, ...)
```

Linux System Call	Emulated System Call
int open (__const char* __file, int __oflag, ...);	int open (__const char* __file, int __oflag, ...);
ssize_t read (int __fd, void *__buf, size_t __nbytes) __wur;	ssize_t read (int __fd, void *__buf, size_t __nbytes) __wur;
int close (int __fd);	int close (int __fd);
int select (int __nfds, fd_set *__restrict __readfds, fd_set *__restrict __writefds, fd_set *__restrict __exceptfds, struct timeval *__restrict __timeout);	int select (int __nfds, fd_set *__restrict __readfds, fd_set *__restrict __writefds, fd_set *__restrict __exceptfds, struct timeval *__restrict __timeout);

Table 5.7: Mapping from system calls to their emulated equivalent

This function tries to open the file `__file`, and returns a file descriptor, an integer, used as a handle subsequently by the application to read data from that file. The `ExecutionEnvironment` abstraction provides an abstraction of a single file, `PFile`, and a collection of these `PFiles`, analagous to a filesystem, can be addressed by a char* key string. When the program running on the IPAQ, invokes the `open(...)` function, it is transparently redirected to the LinuxExecutionEnvironment equivalent

```
int LinuxExecutionEnvironment::open_local(__const char* __file,
                                          int __oflag, ...)
```

This method searches the `ExecutionEnvironment`'s abstraction of the file-system, and returns a file descriptor, linked to a `PFile` instance, for the parameterised file. Subsequent calls by the application to system calls such as `read(...)`, and `write(...)`, that use that file descriptor, will interact with the associated `PFile` instance.

This redirection is achieved using the compile-and-link approach, whereby calls to the system API are intercepted when the program is being compiled and are linked to the equivalent API of the `LinuxExecutionEnvironment`. At the time of the program compilation, the IPAQ's call to include the appropriate definition files, listed in figure 5.24 for their respective system calls are intercepted. PiCSE-specific versions of these header files implement the mapping that are listed in table 5.7 and that were just described.

```
#include <stdio.h> /* Standard input/output definitions */
#include <unistd.h> /* UNIX standard function definitions */
#include <fcntl.h> /* File control definitions */
#include <errno.h> /* Error number definitions */
#include <termios.h> /* POSIX terminal control definitions */
```

Figure 5.24: System call definition files for the invoked system calls

Modelling the scenario's hardware components

This scenario is comprised of four physical components: an environment, "the room" in which the scenario takes place is defined. There are three scenario objects present with that room: the vehicle, its sonar sensor and a detectable obstacle.

The Room_Layer Instantiation and a Detectable Obstacle

The room is captured as a `Room_Layer` which is an instantiation of the `EntityLayer` abstraction. It models a physical space with a granularity of one metre, and that contains a single detectable `Object` instantiations at a fixed Cartesian location (10,10). This layer is queryable by a sensor instantiation.

The Vehicle Instantiation

The vehicle is modelled using a custom extended Object class. This approach was chosen to demonstrate how a specific mobility pattern could be implemented, thus accommodating more mobility-based scenarios in addition to the frequently used random waypoint mobility patterns. In this instance, the vehicle is defined to move from a fixed location in a specified direction and at a fixed speed.

The Sonar_Sensor Instantiation

The `Sonar_Sensor` class is an sub-class instantiation of both the `Sensor` and `Emulated` abstractions. The `Sensor` aspect of the instantiation queries the `Room_Layer` for sensor data. At the time of construction, the sensor object is parameterised with its data source, an instance of the `EnvironmentLayer` abstraction, that models the location of obstacles within a physical

space.

A sensor reading occurs when a sensor object takes a single reading and manipulates some data value returned, object present, into the format required by the emulated interface. In the case of this physical sensor, the actual sensor reading is eleven characters long: the first four characters identifying the sensor; the next six characters provide the reading and the final character, "/n" delineates the sequence.

In order to potentially capture some of the timing characteristics of an actual sensor, this sensor model produces two types of events: CHS_SENSE events and CHAR_ events. Creating a scheduling difference between the time a sensor reading takes place (CHS_SENSE) and when the reading is pushed to the PIC board (CHAR_EVENT) allows a user to model a characteristic of an actual sensor such as a delay. The sensor is additionally parameterised with the frequency of the sensor events, in this case, every 0.5 seconds.

The `ObjectLogger` Instantiation

Finally, an `ObjectLogger` is introduced to monitor the sonar_sensor and record the events of interest for a viewer. The `ObjectLogger` component is parameterised with an instance of an `Object` abstraction and with the file name to which it logs events. By default, no events are logged, so the `ObjectLogger` subscribes to all events by invoking the `Sensor` abstraction's method, `::Attach(this)`.

Interlinking of simulated hardware and software components

Logical Co-location of `ExecutionEnvironment` and `Sensor` Abstractions

The scenario requires that the currently separated instances of the sensor, the application and its execution environment be interconnected in order to achieve its modelling objective. The application should run on the execution environment, and should consume sensor events that are delivered to that execution environment via an emulated PIC board.

Figure 5.25 shows the now familiar instantiation and co-location of an application with its ExecutionEnvironment. More interestingly, figure 5.26 shows the mechanism by which the sensor is attached to its ExecutionEnvironment instance. An instantiation of an `Emulated`

```
LinuxExecutionEnvironment* IPAQ = new LinuxExecutionEnvironment();
ChsApp* ca = new ChsApp(5,LAPTOP);
```

Figure 5.25: Co-location of ExecutionEnvironment and the IPAQ application

```
Sonar* sonar_sensor = new Sonar(rl,IPAQ);
LAPTOP->addEmulatedHardware(sonar_sensor, "/dev/ttyS0");
```

Figure 5.26: Co-location of ExecutionEnvironment and Sensor Objects

abstraction, the sensor in this scenario, must be added to an `ExecutionEnvironment` abstraction in order for any potential interaction between a `Sensor` and an application can occur. This binds the `Emulated` instantiation to a logical point, in this case a `PFile` addressed at the location "/dev/ttyS0" , within an `ExecutionEnvironment`. With this instruction, any events raised by the sensor are pushed to the ExecutionEnvironment's emulated file-system, whereupon the IPAQ application can read and utilise them.

Logical Dependency between `ObjectLogger` and the Sonar_Sensor

A logical dependency is created automatically between the `ObjectLogger` and the `Object` instantiation, which is the `Sonar_Sensor` in this case. The `ObjectLogger` is notified of all events in the case that no particular event type is subscribed to. In the case of the `Sonar_Sensor` instance in this scenario, it raises TRANSFORM events, which are the events raised when a sensor takes a periodic reading. It also raises CHAR_EVENT events, which are generated when a character is pushed to its `Emulated` instantiation. When these events occur, all subscribers, the `ObjectLogger` in this scenario, that are associated with the Sonar_Sensor are notified. On notification, the `ObjectLogger` records the event, and its timestamp within its logfile.

5.4.3 Execution of the modelled domain

Initialising the experimental scenario

The reader should refer to sub-section 5.3.3 of the earlier STEAM scenario for a description of the initialisation of the standard PiCSE components which are created and initialised for every simulation. A single application, executionEnvironment and sensor are instantiated as described in section 5.4.2 and the scenario is now ready to be executed.

Execution of the simulation

The experiment was performed on an off-the-shelf Dell Latitude C400 laptop, equipped with a minimal 512 MB of RAM and running on Ubuntu 6.10, Edgy Eft, an open source operating system built around the Linux kernel.

The experiment was executed within a terminal and figure 5.27 shows an excerpt of the logged output of the experiment. The output lists some scenario parameters at the outset, such as the size of the world and the duration of the experiment. The timestamps denote the different times, in milliseconds, that the listed events occurred at.

1. The first TB_event (Thread Begin) occurs at 5 milliseconds. At this timestamp, the IPAQ program has now begun, and will enter into it's paused state, waiting for a sensor event to awaken it.

2. At 500, the first M and Sonar events occur. This is first scheduled sensor event. The sensor takes a reading at this time and schedules a series of serialisation events, whereby each character of the sensor reading is written to the PFile file-system individually. As there are eleven characters in the sensor reading, eleven E_ events are scheduled at the timestamps 501ms, 502ms, 503ms, etc up to 511ms.

3. At each of the scheduled times, 501ms and 502ms for example, an E_ and a TS_ event occurs. An E_ event is an event of an emulated object, the sensor in this case. The TS event denotes a Thread Switch event. This occurs when the scenario execution switches from the master simulation thread, which drives the entire experiment, to an emulated

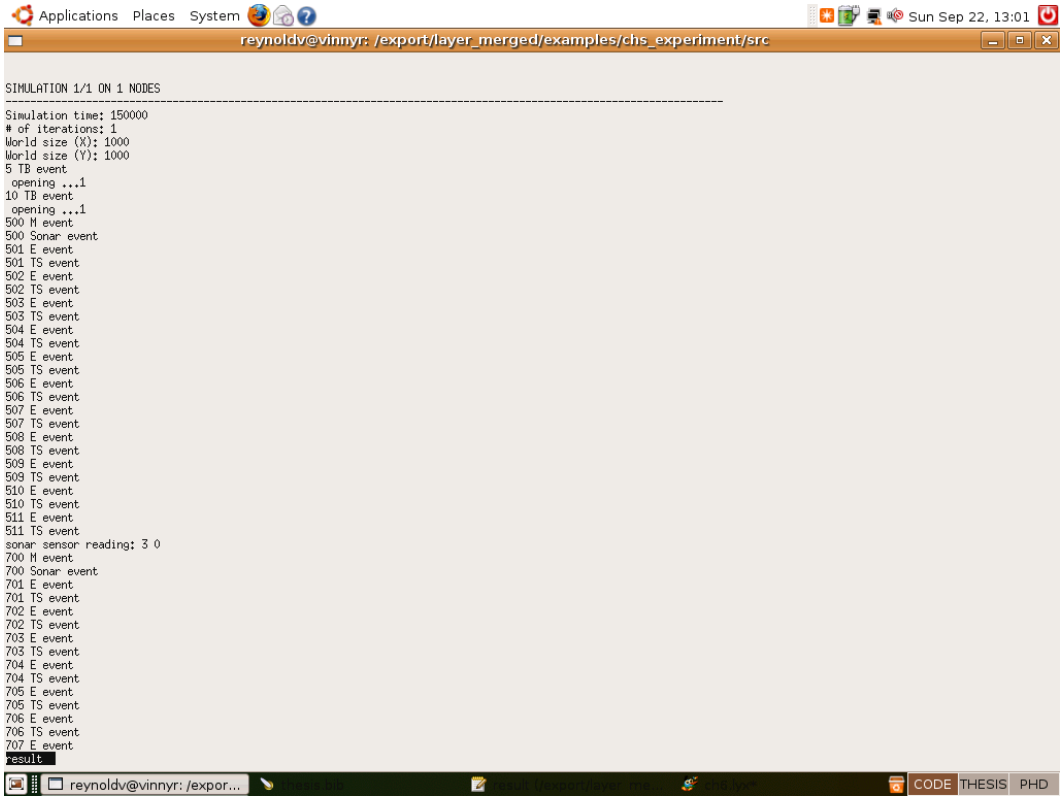

Figure 5.27: A screenshot showing the logged output from the Car Hardware Scenario evaluation.

application thread, in this case the IPAQ application which is reading each individual sensor character.

4. After the final character arrives at 511ms, the application outputs[2] the sensor reading which was been read by the application. In this scenario, the sensor reading is an eleven character sequence of '0', '0', '0', '3', '0', '0', '0', '0', '0', '0', '/n'. The first four characters are to be interpreted as the sensor id, the value 3, and the final six characters denote the actual sensor reading, in this case, the value 0. The application outputs the sensor reading when it has read the de-limiter character which it interprets as the end of the sequence.

5.4.4 Scenario Analysis

Validation of the scenarios requirements.

The scenario introduction outlined two requirements in order for the particular scenario requirements to be validated in addition to the PiCSE evaluation requirements. The first requirement, "Emulation of system call based programming interfaces", is achieved by the provision of the IPAQ application which is built upon five system calls which have been listed in table 5.7.

The second requirement, that the interaction between the simulated and emulated components is captured realistically. By realistically, it is meant that the emulated components produce the same outputs as a real sensor, and that any interaction with an independent application is the same as the original sensor upon which it is modelled.

The outputs of the sensor have been successfully replicated in the simulation of this scenario. Evidence of this is provided by the unmodified application code implemented by the IPAQ application. The compile-and-link approach utilised by the PiCSE framework means that no modifications are required to run the IPAQ application within the framework. The applications invocations of the underlying system calls are transparently redirected into the

[2]The original IPAQ application did not produce this output but the program was slightly modified in order for the logged experimental output to have some meaningful data and provide some insight into the internal workings of the application.

PiCSE framework where the expected behaviour is simulated using the analogous functionality provided by the PiCSE core.

Insights

Although this scenario is rather simple in terms of the number of the components that have been modelled, the modelling of the interaction between the sensor and the application is realistic. The scenario does not place much emphasis on realistic modelling of the physical aspects of the sensors actual sensing characteristics, so no conclusions can be drawn regarding the frameworks modelling of hardware components.

The compile-and-link approach utilised again in this scenario has again worked successfully as a second class of application has been successfully emulated. The integration of the `LinuxEmulated` interface, and the `LinuxExecutionEnvironment` class enable the emulated interaction between modelled hardware and software components. If, theoretically, the `Sonar_Sensor` was to simulate a hardware failure, and fail to send any further characters to the `ExecutionEnvironment`, the correct application behaviour would still occur: the program would wait indefinitely until the next character arrives! The simulation of the sensor's behaviour is achieved at a high level of granularity. Individual character events are simulated and the scenario can even simulate a delay on the writing to the emulated file-system. The accurate modelling of the interaction is the key feature of this scenario, and in this respect, the framework meets the modelling requirement.

Finally, one logical dependency was used in this scenario. The `ObjectLogger` component used PiCSE's built-in support for logical dependencies, to subscribe to events generated by an `Object`, in this scenario, the Sonar sensor. This class of logical dependency is independent of the instantiations of this scenario, and this independence offers further validity to the PiCSE framework's claim of flexibility and composability.

Evaluating the PiCSE requirements

The thesis's requirements are measured for this scenario with respect to their associated metrics and presented in table 5.8 .

Metrics	Car Hardware scenario
The number of diverse instances of all hardware components	1 environment (room), 2 objects (car,sensor)
The number of instances of emulated components, both applications and middlewares.	1 application (IPAQ), 1 ExecutionEnvironment (Linux)
The number of instances of component inter-relationships, which is defined as an interaction between two separately created instances of PiCSE abstractions.	1 object-environment relationship, 1 application-executionEnvironment relationship, 2 application-object relationship (IPAQ-sensor, IPAQ-car)
The number of instances of inherited reusable abstractions. Measure the proportion of the simulation that is comprised of re-usable domain specific elements.	1 executionEnvironment (Linux) 31.6%[1]
The proportion of the simulation that is provided by the PiCSE frameworks core classes.	65.5%
Demonstration of network functionality in application scenarios.	n/a
Demonstration of evidence of evaluation support functionality.	TRUE (1 ObjectLogger)

Table 5.8: Requirements analysis for the Car Hardware scenario

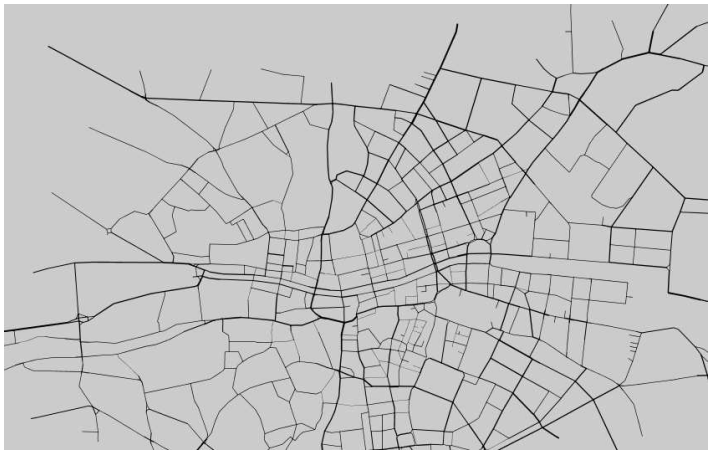

Figure 5.28: The Dublin City road traffic network

The Car Hardware Scenario provides a second demonstration of the functionality provided by the PiCSE framework. A second class of application, those built upon low-level system calls have been emulated successfully, while the common pervasive interactions between hardware,

A final over-arching analysis of the requirement satisfaction across all three scenarios is completed in the final section of this chapter. software and environmental components have been demonstrated again. It is noted in the table however, that the comparatively low percentage, 31.6%, of the proportion of simulation comprised of re-usable domain specific elements is accounted for by the implementation of the emulated Linux environment. A high proportion of this implementation is related to low-level functionality such as thread control and read-write operations and a lot of this functionality is provided from within the PiCSE core and not from within the actual emulated environment which was created for this scenario.

5.5 The Intelligent Transportation Systems Scenario

The third and final instantiation of the PiCSE framework is a simulation of an Intelligent Transportation Systems (ITS) scenario. The ITS scenario is a large-scale simulation of a portion of the Dublin City road network, shown in figure 5.8 that is approximately 5 km^2 in area and the number of vehicles being simulated at any one time is in the order of thousands. ITS scenario's are particularly suited to simulation because of the inherent difficulty, cost and

risk in evaluating the scenarios in the real world.

ITS scenarios are considered to be part of the pervasive computing domain, and contain many of the characteristics of typical pervasive computing scenarios such as handling mobility for users and reactive intelligent physical spaces and artefacts. The scenario has certain properties that exercise elements of the PiCSE framework. There are a large number of dependencies, such as the relationship of vehicles with each other and with other features of the environment such as the traffic lights. The ITS scenario is both inherently dynamic and large scale and so this scenario tests those aspects of the framework as well. Although there are no emulated applications within this scenario, there is control logic in place that models the behaviour of the traffic lights.

5.5.1 Scenario Modelling Requirements

The primary motivation for choosing the ITS scenario was both its scale and its complexity. A working simulation of thousands of interacting instantiations would provide an excellent indicator of the frameworks capacity to simulate large scale pervasive computing scenario, whilst the variety and complexity of the features of an ITS scenario will test the abstractions that the framework provides.

The scenario is comprised of a road network of 270 junctions (nodes) and 459 roads (edges) linking those junctions. Over a simulated two hour period, over 10,000 simulated vehicles traverse that road network, each travelling according to its own pre-defined paths and at any one time, there is approximately 1000 simulated vehicles within the road traffic network. In comparison to the first two scenarios evaluated for this thesis and and to other pervasive computing application domains this is undeniably a large scale scenario.

The complexity of the scenario is reflected not just in the types of components required to make up the scenario but also in their internal complexity and dependence on other instance types for both their implementation and behaviour. A rich environment model must be instantiated capturing both static and dynamic aspects of the environment. A model of the road topology is required that reflects the normal constraints of what we consider to be a traffic system, such as the number of lanes, and any speed limit that may exist on that road.

A dynamic aspect of the environment is the modelling of the location of all of the vehicles.

For example, inductive loop sensors that report the detection of vehicles within the local road network. Traffic light actuators are present at the incoming road at a junctions within the system. They actuate indirectly on the environment, by influencing a vehicle's behaviour. Finally, the movement pattern of the vehicles (users) must also be modelled, taking into account factors such as local "rules of the road" and the position and behaviour of other vehicles amongst others.

Finally, messages are exchanged wirelessly between the traffic light controllers, so the instantiation must track the location of thousands of both mobile and static, senders and receivers. Although there is internal complexity and a large amount of control logic within the behavioural aspect of a simulated aspect, the scenario does not contain any emulated applications.

5.5.2 Building the Model

Modelling the Scenario's Software Components

There are no emulated applications or middlewares within this scenario. Any control logic that governs behaviour within any of the scenario's physical components such as the vehicles or the traffic lights are described inline as part of the next section.

Modelling the scenario's Hardware Components

Modelling of Road Network Environment

The road topology is initially described in a custom XML file format, which is parsed and then instantiated to create the class objects that are used in the simulation. In the example of the topology definition that is shown in figure 5.29 , a junction with the unique ID 1526 is defined as a junction containing a traffic light (TL), and is located at the Cartesian location [379278.0, 266013.0]. An "incomingJunction" is defined, which is the definition of an road (an edge) that can carry traffic from that junction (1525) into the junction 1526. The number of lanes (2), the length of the road (141.7m), and the max speed on the road (30 km/h)

```
<junctionRec>
  <id>1526</id>
  <type>TL</type>
  <location>
    <xCoordinate>379278.0</xCoordinate>
    <yCoordinate>266013.0</yCoordinate>
  </location>
  <incomingJunction>
    <id>1525</id>
    <numLanes>2</numLanes>
    <linkDistance>141.69333082400175</linkDistance>
    <maxVelocity>30.0</maxVelocity>
    <outgoingJunctionRef>
      <id>1527</id>
      <actionType>R</actionType>
    </outgoingJunctionRef>
    <outgoingJunctionRef>
      <id>850</id>
      <actionType>S</actionType>
    </outgoingJunctionRef>
  </incomingJunction>
  ...
```

Figure 5.29: A snippet from the citycentre.xml file describing the road network topology and constraints.

are all defined for the incoming road. The final part of the incoming road definition is the list of "outgoingJunctionRef"s. These are the legitimate exits that a vehicle can make when approaching the junction 1526 while approaching from junction 1525. There are two possible exits for the 1525-1526 road segment: a right (R) turn onto the road segment 1526-1527, and a straight turn onto the road segment 1526-850. The complete XML definition of junction 1526 can be found in the appendix.

An XML parser is used to parse and instantiate a series of Junction_Objects that model the corresponding JunctionRecord from the XML file. An extension of the PiCSE Object class, each junction can potentially schedule and create events, but in this scenario, they are simple conceptual objects that have a physical location, a unique junction identifier and references to other connected junctions.

Each instance of the JunctionObject class is instantiated and stored in an instantiation of an `EntityLayer` extension called the `RoadNetwork_EntityLayer` that captures the modelling of the static topology of the road network. The `RoadNetwork_EntityLayer` class's key-based API is used by other models to query aspects of the road network. For example, a vehicle that has to explore the path ahead, can use the current junction ID as the key, and can use the returned `Junction_Object` to explore connected junctions and hence determine available paths.

Modelling of Car Behaviour

Each vehicle within the scenario is modelled as an instance of the `Vehicle_Object` class, an extension of the Mobility_Object class. Each `Vehicle_Object` is responsible for updating its own position. A detailed description of the algorithm employed here is outside of the scope of this thesis, but the inputs on which the algorithm makes it control decisions are based on scenario variables that are modelled so these are discussed.

Based on a car-following model, the algorithm takes the following factors into account:

- Status of traffic lights on the current road
- The location and status of other vehicles on that road and upcoming adjoining roads

```
10519,1,1,16,1169,1153,1442,984,1590,702,1589,1586,1266z,1444,1636,1483
,1701,1622,1623,610
```

Figure 5.30: A sample entry from the path definition file

- The topology of the road itself and its local constraints
- The vehicles own pre-defined path.

Traffic Light Status The status of a traffic light object is determined by querying the TrafficLightEntityLayer object using the road segment id. This returns a trafficLight object which contains the traffic light data. Traffic light timing information is available, but only the enumerated values of red, green, and amber are used by the vehicles behavioural algorithm.

Location of other vehicles The modelling of the entirety of the vehicles within the network is captured by another instantiation of the `EntityLayer` abstraction, the `VehicleEntityLayer` class. Its key-based API is used to query the EntityLayer for all vehicles on a particular road segment identified by the concatenated string of two junction IDs. The returned vehicleObjects are iterated over by the vehicle's behaviour algorithm to determine the vehicles that are nearby and their

Road Topology Variables relating to a road segment that a vehicle is travelling on, such as the number of lanes, any local speed restrictions, and its length are all taken into consideration in the vehicles behavioural algorithm. The same factors for any outgoing roads beyond the end of the current road segment are also taken into account.

Vehicle Paths Finally, the behavioural algorithm considers the vehicles pre-determined path. The vehicle path is defined as a series of junctions that the vehicle must traverse through the road network and the path begins and ends at a leaf node within the road network. Figure 5.30 shows the path definition for a single vehicle.

1. The first value, 10519, is the timestamp(ms), that the vehicle will begin its journey.

2. The second value, 1, indicates the vehicle type. A "1" denotes a car.

3. The third value denotes the driver type, allowing for future types of driver behaviour to parameterised.

4. The values from position four until the end are a series of junction IDs that the vehicle must pass through to complete its path. The first road segment will then have the ID "16-1169", and will then turn onto the road segment "1169-1153".

In addition to the current road segment, the behavioural algorithm takes into account the upcoming road segment that a vehicle will enter and uses that path information to determine the correct lane position for executing a turn onto that road segment.

Modelling of Traffic Light Behaviour

The traffic light model is implemented as combination of a controller and actuators. The `TrafficLightController` implements a fixed-phase cycle algorithm in order to control the actions of the `ApproachLights_Actuator` that exists on each junction. A fixed-phase cycle algorithm is the traffic light control algorithm widely used today, whereby a fixed time period is defined for each of the traffic lights three possible states, red, green or amber.

The `ApproachLights_Actuator` instances are added to `TrafficLightController` at the start of the framework's instantiation. The `Actuators` have an enumerated state, that is queried by vehicles and is of the governing factors in the vehicle's behavioural algorithm.

5.5.3 Execution of the modelled domain

Initialising the Experimental Scenario

Before the simulated scenario is executed, two external data sources are parsed and loaded into the experiment.

The definition of the vehicle's paths is contained in a file "data/TraceFile_wholemap_0", which contains 10,008 vehicle tracepaths for a simulated two hour period. Figure 5.31 displays a code snippet demonstrating how the simulator handles so many potential event generators.

```
// define the trace source file
string sources[noOfSources] = { "data/TraceFile_wholemap_0" };
  // Create Parsers for the trace files and schedule their first events.
  for (int s=0; s<noOfSources; s++) {
    // parse the file and get the first row-entry
    LineParser* a = new LineParser(sources[s],&network_map);
    long int t = a->getNextTime();
    if (t!=-1) {
      // schedule the next car load event
      Simulator::schedule(&generate_cars,GENERATE_CARS,t,a);
    }
  }
```

Figure 5.31: A section of code demonstrating how the car path file is parsed.

```
RoadNetworkEntityLayer network_map;
junctionParser jP(argv[ARG_JUNCTION_FILE], &network_map);
jP.processXML() ;
```

Figure 5.32: Parsing the CityCentre.xml map file

Events are scheduled and created for the next vehicle and path only that are contained within the file. When that timestamp is complete and that event has been executed, the following entry is read and the appropriate event is scheduled. Hence, the number of GENERATE_CAR events that are scheduled is never more than one, and not 10,008.

The definition of the road topology is defined in "data/mapFiles/citycentre.xml" and parsed and loaded into the EntityLayer as shown in figure 5.32 . The junction parser extracts the junction ID, and the junctions outgoing and incoming junctions, and from those constructs the appropriate road segment instances and the vehicleEntityLayers that will store vehicle objects that are on those segments. The experiment is now ready to execute.

Execution of the simulation

Figure 5.33 show the outputs from the experiments that have been loaded into a viewer made for this particular scenario. The framework logs the locations of the vehicles and the status

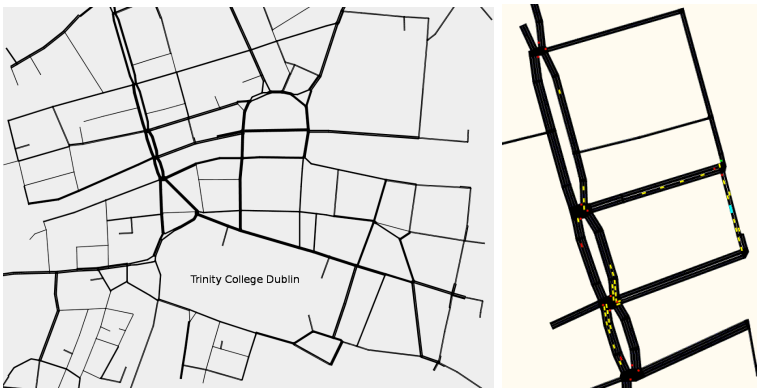

Figure 5.33: Two screenshots from a visualisation of the Dublin traffic scenario

of traffic lights at each timestep. The first screenshot shows a zoomed out partial of the map defined by the citycentre.xml file. Irish readers will note that the map centred around the city centre of Dublin. The second screenshot is a close up from the upper left quadrant area of the map and represents and approximate 750m long section of road[3].

The vehicles are denoted as yellow rectangles, whilst the traffic lights which are located and intersections between road segments are shown as small circles with either read, green or amber.

5.5.4 Scenario Analysis

Insights

Observed Vehicle Behaviour

Vehicles follow a rather recognisable pattern of starting slowly and respect the distance and speed of vehicles in front of them and even demonstrate some excellent lane-changing behaviour in anticipation of upcoming junction traversals. At a macro level, the vehicles are seen to bunch up at traffic lights and at times of traffic congestion. Simulated accidents can be scheduled within the scenario. However, as the vehicles behavioural algorithm does not take accidents and path re-routing into account, a rather predictable blockage and reduced throughput inevitably occurs.

[3]From O'Connell Bridge at the bottom up along O'Connell St!

Handling The Scenario Complexity

The most difficult aspect of the ITS scenario was to model the complexity of the scenario and its components. Figure 5.34 shows a sequence diagram showing the key interactions between the `Vehicle_Object`, and other simulated `Object` and `Layer` instances. All of these interactions are supported by the underlying abstractions, `EntityLayer` and `Actuator`. The ITS environment is a complex layered model where EntityLayer instantiations, representing traffic, the road network itself, and traffic light actuators interact, sometimes explicitly to model a complex scenario.

The static topology of the road network does not lend itself to the location-based model provided by the `EnvironmentLayer` abstraction. A location, in this instance would only refer to a point on the road, whereas a logical identifier, such as roadID can refer directly to the road itself. For this reason, the `EntityLayer` abstraction and its key-based API are better suited to the modelling of the road network. For similar reasons, a key-based API is better suited to modelling the location of the vehicles themselves. In this scenario, the only interaction between vehicles was the implicit interaction that governed the vehicle's behaviour. This particular scenario did not use the location-based API of the `EntityLayer`, but a simple hypothetical extension of the scenario validates the `EntityLayer`'s support for both a key- and location-based API.

In total, over ten thousand additional lines of code were required to capture the scenario's complexity. A large proportion of this was in the modelling the cars behavioural algorithm, a particularly complex algorithm that had to take a number of input variables from other parts of the scenario as well as calculating a large number of inline variables before deciding on the optimum car behaviour. The model of the Vehicle_Object's behaviour was underpinned by the availability of the detailed model of the physical ITS environment and its components. Therefore, the flexibility of the framework's abstractions, and the support for building complex inter-dependent abstractions is validated.

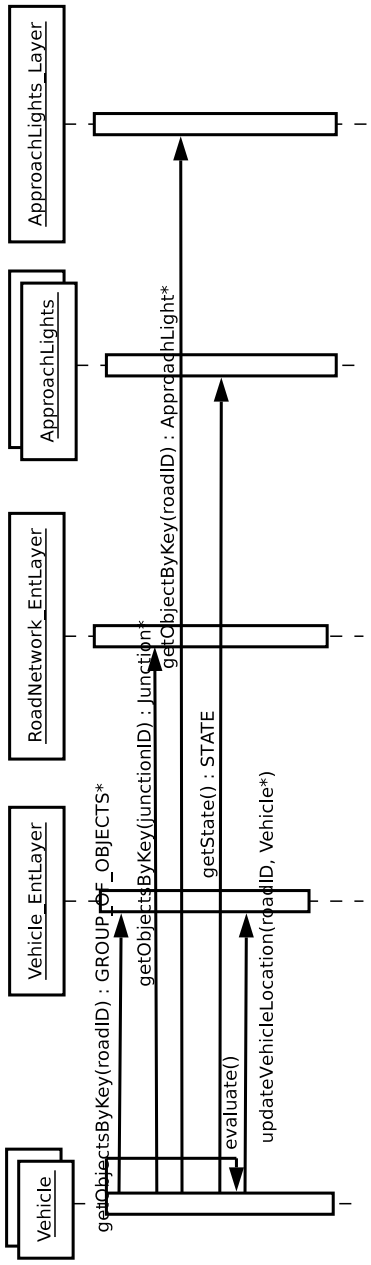

Figure 5.34: A vehicle interacts with its environment to inform its next movement

Handling Large Scale Scenarios

Addressing the requirement identified that was to support large-scale applications, the ITS scenario successfully modelled a large number of vehicles interacting, and a large, albeit simplified model of an ITS wireless network. This is due in part to the physical and logical partitioning employed with the `EntityLayer` abstraction. The physically partitioned `EntityLayer`, where nearby `Vehicle_Objects` are grouped together, and can be addressed by location, reduces the effort required by the network simulator to calculate the number of wireless receivers.

The logical partitioning of the space groups `Vehicle_Objects` into sets based on their location within the road network. This ensures that `Vehicle_Objects` only interact with those on the same road, and not those that are physically close by but on separate roads.

However, the framework's support for modelling large scale scenarios, and this scenario in particular, has not been measured. Therefore, there is no quantitative analysis of PiCSE's support for large-scale scenarios, other than to state that in this scenario, up to 1000 interacting vehicles and other simulated components were present at any time with the scenario's simulated duration.

Evaluating the PiCSE requirements

Table 5.9 presents the requirements analysis of this ITS scenario and measures the frameworks seven requirements against the identified metrics. Of note in this scenario are the number of instances of diverse instances, 8, and the number of inter-relationships, over 20, identified in table 5.9. These numbers strongly support the argument that this scenarios complexity is a validation of the framework's capacity to model heterogeneous pervasive computing scenarios.

A large effort was required to implement this scenario. Over ten thousand ITS specific lines of code were required to implement the behavioural algorithms and modelling complexity that formed this scenario, however a large part of that of that codebase is made of re-usable components that can be used in future ITS simulations and is now in the form of an ITS PiCSE sub-framework. The complexity required to model the scenario's behaviour again is reflected in the relatively low ratio (24%) between the PiCSE core classes and the overall scenario implementation. The implications of this are now discussed in the overall requirements

Metrics	ITS scenario
The number of diverse instances of all hardware components	3*objects instances (car, traffic light, junction), 4*entityLayers (JunctionLayer, VehicleEntity-Layer,TrafficLightEntityLayer, 1* actuator (TL_actuator)
The number of instances of emulated components, both applications and middlewares.	n/a
The number of instances of component inter-relationships, which is defined as an interaction between two separately created instances of PiCSE abstractions.	20+
The number of instances of inherited reusable abstractions. Measure the proportion of the simulation that is comprised of re-usable domain specific elements.	4 primary (Car_Object, TL_Object, Junction_Object, RoadEnvironmentLayer) 57.7% of 11,128 LOC
The proportion of the simulation that is provided by the PiCSE frameworks core classes.	24% of 14,659 LOC
Demonstration of network functionality in application scenarios.	n/a
Demonstration of evidence of evaluation support functionality.	ITS scenario

Table 5.9: Requirements analysis for the ITS scenario

validation and the chapter's conclusion.

5.6 Requirements Validation and Conclusion

This chapter presented three distinct instantiations of the PiCSE framework. Overall, these instantiations exercised the framework's flexibility and extensibility through the implementation of a diverse range of simulation and emulation modelling requirements that were driven by the scenario's requirements. Although the requirements were met by the architectural design and implementation of the framework, the purpose of these three scenarios was to validate the approach and hypothesis of the thesis.

5.6.1 Requirements R1, R2, and R3

The modelling of both hardware and software components is provided by the high-level classes, `Sensor`, `Actuation`, `EnvironmentLayer`, `EntityLayer`, `ApplicationWrapper` and `ExecutionEnvironment` and their respective inherited classes. PiCSE's support for modelling multiple simulated heterogeneous hardware and software elements and the framework's flexible implementation allows the potential interaction, combination and inter-dependence of these components without restriction. The frameworks support for this is demonstrably clear in table 5.10 which outlines the many scenario instances that were created using the PiCSE framework

Table 5.11 , summarises the implementations of the key abstractions that were identified. The three scenarios necessitated a diverse number of instantiations of the abstractions, both software and hardware, that are supported by the PiCSE framework. In addition, as seen in the scenario descriptions there were many dependencies captured as interactions between these instantiations.

Overall, the flexible instantiation of multiple abstractions, of both software and hardware elements, verifies the framework's meeting of requirements R1, R2, and R3 and these have been met by the architecture, design and implementation of the framework.

Metrics	STEAM Emulation scenario	Car Hardware scenario	ITS scenario
The number of diverse instances of all hardware components	1 object type, 1 environment type	1 environment (room), 2 objects (car,sensor)	3*objects instances (car, traffic light, junction), 4*entityLayers (JunctionLayer, VehicleEntity-Layer,TrafficLightEntityLayer, 1* actuator (TL_actuator)
The number of instances of emulated components, both applications and middlewares.	2 applications, 1 middleware instance	1 application (IPAQ), 1 ExecutionEnvironment (Linux)	n/a
The number of instances of component inter-relationships, which is defined as an interaction between two separately created instances of PiCSE abstractions.	1 object-environment relationship, 1 application-middleware relationship, 1 application-object relationship	1 object-environment relationship, 1 application-executionEnvironment relationship, 2 application-object relationship (IPAQ-sensor, IPAQ-car)	20+

Table 5.10: Overall requirements analysis for R1, R2 and R3

		STEAM Emulation	Car Hardware Emulation	Dublin Traffic Simulation
Hardware Flexibility	Sensors	n/a	•	•
	Actuators	n/a	n/a	•
	Environment	○	•	•
	Mobile nodes	○	○	•
Software Flexibility	Middleware	•	n/a	n/a
	HW Component	n/a	•	n/a

Table 5.11: The hardware and software models required by the three evaluation scenarios

5.6.2 Requirement R4

The support of the PiCSE framework's architecture to address the requirement of extensibility is met by two aspects of its design. An instantiation of the PiCSE framework can itself be a domain-specific sub-frameworks, composed of extensions of the software and hardware abstractions provided by the *PC_Abstractions* class category. Additionally, the *Abstract_Interfaces* class category provides a set of interfaces that allow the development of new abstractions that are not currently modelled within the existing framework implementation. These extensions can interact and interoperate with the existing *PiCSE_Core* and the existing abstractions. Additionally, the split-level design of the framework's emulation component allows new emulation application APIs to be added when in the future when new platforms or middlewares emerge.

Table 5.12 demonstrates the number of lines of code and the instances from the three scenarios that could be re-used in future simulations. Only one of the scenarios, the Dublin traffic scenario, was implemented as an intelligent transportation systems (ITS) sub-framework, although all three instantiations could be extended if required. The traffic simulator has subsequently been extended to implement a range of traffic light controller algorithms, and evaluate vehicle coordination and planning algorithms. As there is only one instance of a

Metrics	STEAM Emulation scenario	Car Hardware scenario	ITS scenario
The number of instances of inherited reusable abstractions.	1 middleware instance	1 executionEnvironment (Linux)	4 primary (Car_Object, TL_Object, Junction_Object, RoadEnvironmentLayer)
Measure the proportion of the simulation that is comprised of re-usable domain specific elements.	53.1% of 898 LOC	31.6%[1]of 1856 LOC	57.7% of 11,128 LOC

Table 5.12: Overall requirements analysis for R4

Metrics	STEAM Emulation scenario	Car Hardware scenario	ITS scenario
The proportion of the simulation that is provided by the PiCSE frameworks core classes.	79.7% of 4429 LOC	65.5% of 5387 LOC	24% of 14,659 LOC

Table 5.13: Overall requirements analysis for Requirement 5.

sub-framework, this requirement is only partially validated. These functionalities and design features meet the framework's extensibility requirement, R4.

5.6.3 Requirement R5

The *PiCSE_Core* framework engine comprises of a set of components that are common to all PiCSE simulations. The *PiCSE_Core* engine was re-used without modification across all three scenarios and table 5.13 lists the percentage of the scenario simulation that was provided by the PiCSE core. The wide variance in the percentages listed is in inverse proportion to the complexity of the scenario, but what the figures clearly show is that for small scale scenarios with a limited numbers of modelled components, PiCSE arguably provides a significant proportion of the foundations for the development of those simulations.

This reusable engine, and the identifying of recurring software patterns that are captured

Metrics	STEAM Emulation scenario	Car Hardware scenario	ITS scenario
Demonstration of evidence of evaluation support functionality.	FALSE	TRUE (1 ObjectLogger)	Yes. Logging and scenario log viewer.

Table 5.14: Overall requirements analysis for Requirement 7.

within the classes for the hardware and software elements, and their inter-dependencies meets the framework's reusability requirement, R5.

5.6.4 Requirement R6

The PiCSE framework provides two feature that address the aspect of network simulation within pervasive computing scenarios. The `Net_`Interface class and the underlying Network Manager component provide a rudimentary API for the simulation of network behaviour. The STEAM emulation scenario and the Dublin traffic scenario both utilise the basic network simulation component. Both scenario's are comprised of both static and mobile nodes, and the traffic scenario has implemented but not demonstrated both wired and wireless communication channels.

Networked communication between ITS components is frequently posited as a means of augmenting and improving a drivers experience whilst driving, and as a means of maximising vehicle throughput through a road network. The ITS PiCSE scenario accommodates these scenarios by extending the scenario's components with the Net_Interface class enabling the exchange of messages across simulated wired and wireless networks.

These aspects of the scenario verify the framework's basic network simulation functionality. and this requirement, R6, has been validated.

5.6.5 Requirement R7

The PiCSE framework's provides a flexible API that allows a developer to specify time windows, where the the behaviour and state of objects of interest within a simulation is recorded.

As shown in table 5.13 , two of the three scenarios utilised experimental logging during

their execution, whether to verify the correct functioning of the STEAM event middleware, or to log and subsequently view the correct vehicle behaviour. The implementation of this logging using PiCSE's API validate the framework's requirement R7.

5.6.6 Conclusion

All requirements which address the core challenges identified in chapter 1 have been fully met in the design of the architecture and implementation of the PiCSE framework. The scenarios presented in this chapter are drawn from a broad spread of pervasive computing application domains, and present a wide range of individual requirements that the framework demonstrably meets.

The framework's elegance in handling the diversity of these scenarios will still providing a meaningful and extensible codebase from which to develop future simulations in other domains is validation of its approach, architecture and implementation. Furthermore it is a validation of the hypothesis of this thesis that it is possible to create such a framework for the simulation of pervasive computing scenarios.

Chapter 6

Conclusions

This thesis presented the architecture, implementation and evaluation of PiCSE, a framework whose abstractions can be instantiated to create simulations of, and simulators for, pervasive computing applications. This chapter concludes the thesis by restating the challenges in creating a generic approach to this task, and presents the main contributions of this thesis in addressing those challenges. The contribution of the thesis is positioned with respect to the state of the art, and finally, potential future work is explored.

6.1 The Challenge Restated

Simulation can provide a quick and cost-effective evaluation tool, that addresses some of the difficulties in evaluating pervasive computing applications. There have been several attempts to provide a generic tool supporting the simulation of this domain; however, they have only met with limited success. Previous work in this area has largely focused on developing models of the physical environment, and emulating applications that exist within that environment. The modelling of hardware devices, such as sensors and actuators, that exist within those environments, and are an integral part of those environments, has been largely isolated. Approaches to building these generic tools have been centred around the extension of existing established simulators of sub-domains of pervasive computing such as wireless sensor networking, or graphical environmental simulators. However, these approaches have been

demonstrated to be insufficiently flexible in being able to integrate aspects of other pervasive computing sub-domains in a generic manner. The open research challenge in this field is to investigate whether a generic simulator can be developed that can model a broad range of existing pervasive computing applications as well as new applications that may emerge in the future.

6.2 The Contribution

The multi-disciplinary and evolving nature of the domain has necessitated the development of a more flexible and extensible approach to the building of these tools, an approach that captures and reflects recurring features and patterns of pervasive computing applications. The contribution of this thesis is the validated definition and architecture of a framework-based methodology for the modelling of pervasive computing scenarios.

The PiCSE framework adopts a bottom-up approach to addressing the modelling of pervasive computing scenarios whereby flexibility, extensibility, and reusability are prioritised as core features of the framework's architecture. In the examination of the state of the art, recurring themes were identified that were common to many pervasive computing scenarios. These recurring themes, such as common devices, recurring interaction patterns were simplified from their implementations, and fundamental abstractions were derived and captured within the framework to implement these themes. The contribution of the framework has been implemented as three class categories, each of which addresses one of the core challenges identified. The *PC_ Abstraction* class category captures a recurring set of abstractions, such as sensors, actuators, the environment, and applications, that may be freely combined in a flexible and heterogeneous manner to create simulations of complex application scenarios. The *Abstract_ Interfaces* class category provides the definition of a set of fundamental abstractions that can be extended to accommodate new application areas that may emerge in the pervasive computing domain. Finally, the *PiCSE_ Core* class category provides a common base upon which all instances of the framework are built and its generic architecture and implementation addresses the challenge of reusability.

This approach contrasts directly with the work of many of the simulators presented in

chapter 2. For example, with the exception of UbiREAL, the simulators that choose to implement models of the physical environment have all done so using 3-d modelling tools. PiCSE's approach, which is similar to that of UbiREAL, begins at the bottom, enabling the modelling of the most simple and basic measurable phenomena. If more complex environments are required, then arbitrarily complex environments can be built from the abstractions that capture these phenomena. In contrast to UbiREAL, PiCSE adopts this philosophy in the implementation of all of its abstractions, whereas UbiREAL only implements its environment in this manner.

Three instantiations of the framework were presented in chapter 5, that were representative of the diverse range of typical pervasive computing applications. These scenarios varied in their requirements, scale and complexity. From simple objects moving in pre-defined mobility patterns, to vehicles that implement complex behavioural patterns, and from emulating message-oriented middlewares to sensor-driven embedded applications, the abstractions supporting these models have proven to be sufficient in enabling the modelling of many pervasive computing domains. It is therefore reasonable to conclude that using a framework is a suitable approach that meets the requirements of modelling a wide range of pervasive computing domains.

6.3 Lessons Learned

Some limitations to the approach chosen have been identified and exposed through the evaluation scenarios. The limitations of using a single machine are perhaps not so important these days with cheaper hardware and pay as you go solutions in the cloud now growing in popularity but the approach utilised in this thesis has placed limitations on the number and complexity of applications that can be simulated at any one time. Whilst, the compile-and-link approach to emulating applications proved feasible for the evaluation of this thesis, a more robust architecture utilising virtual machines would be required to provide a really flexible approach to application emulation. This is arguably a limitation of the implementation of the framework rather than a material flaw within the approach of using a framework itself.

With respect to the evaluation and validation of the thesis, it has been observed that

as the complexity of a simulated scenario grows, the more domain specific knowledge and modelling is required. As the ITS scenario demonstrates, the PiCSE framework can simplify the modelling of the components, it can provide a skeleton around which a developer can bring depth and complexity, but it is up to that user to provide that domain specific knowledge. The PiCSE framework only promises a framework and can give a user with the inputs, however, domain-specific behavioural logic and complexity can require a lot of additional work to bring a scenario to life. There is only so much a framework can provide!

6.4 Future Work

There are numerous potential directions that this work could take in the future. At present, there are limitations within the current implementation that could be addressed in the short term.

The accurate modelling of failure has been an integral part of simulation for wireless sensor networks for a long time, however this feature has yet to cross over into the pervasive computing simulator domain. None of the seven simulators explored in chapter 2 have addressed this issue, but its successful integration is an important requirement for the future. Building robust algorithms, for example inference engines, that are fault tolerant and robust in the presence of both error-prone and inaccurate hardware, is an ongoing area of research within the pervasive computing community. At present, there is no support for modelling the failure that is often present in pervasive computing domains. Modelling failure at the software level would be a welcome development. PiCSE's thread manager component provides rudimentary support for "killing" an application, however, in order to continue with the framework's theme of flexibility, the ExecutionEnvironment and ApplicationWrapper abstractions should be extended to supporting the varying degrees of failure. The second short-term goal is the extension of the framework's emulation capabilities to support multi-threaded applications and to provide bindings for other programming languages beyond C++ so that application code from other applications might be integrated.

The PiCSE ITS sub-framework has been used a simulation tool for a variety of application domains within the intelligent transportation systems research field. Research investigating

inter-traffic light communication as a means of maximising vehicle throughput by coordinating traffic light behaviour has been carried out by several authors including (Salkham, 2010) and (Dusparic 2009).

It could be argued that in these heady days of cloud computing, and the relatively cheap virtualisable hardware providers such as Amazon, that the days of having a simulator or a simulator framework that can run in your local desktop machine or a server are numbered. Whilst there will always be a case for outsourcing computing power and and frameworks that may run upon them, there still exists a niche requirement for a simulation tool that is research focused and allows the rapid prototyping of experiments and scenarios. Access to testbeds is also increasing but again doesn't offer the immediacy of a local machine, that doesn't require scheduled access. For how long this remains the case is open to debate.

In the longer term, the future work that will emerge will depend on the direction that pervasive computing moves in. Five years ago, many pervasive computing systems were based upon embedded sensors communicating using wireless technologies, yet right now, new avenues of research within pervasive computing such as social-sensing, and pervasive health-care are emerging due to the proliferation of smart phones such as the Apple IPhone. The decreasing cost of these devices and wider network connectivity are ushering in a new aspect of pervasive computing where the power and control of these sensing devices rests with the user and not just in the physical spaces occupied by those users. The smart phone's increased ability to contextualise and understand its environment and its ability to integrate with services both locally and on the web make it is difficult to predict what new application areas will emerge in the future within pervasive computing, however the PiCSE framework's flexible and extensible architecture will be at the centre of evaluating those application areas.

Appendix A

Appendix

A.1 PiCSE Architecture Header Files

FigureA.1 gives an overview of the PiCSE architecture and can be used when considering the source header files included below. The source files included below are drawn from the architecture's three class categories, which are described in chapters 3 and 4. FigureA.2 outlines the distribution of the header files across the three class categories.

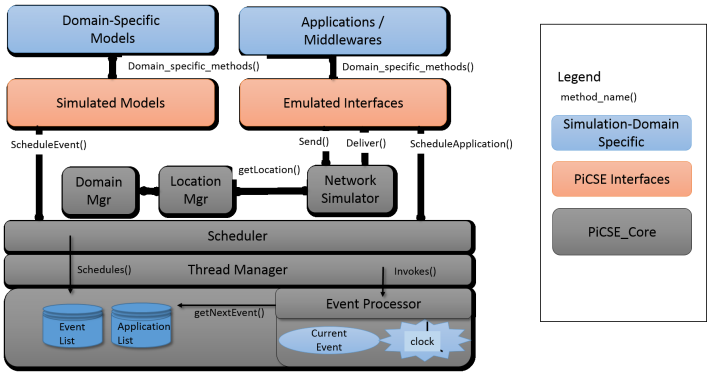

Figure A.1: Architecture Diagram for typical instantiation

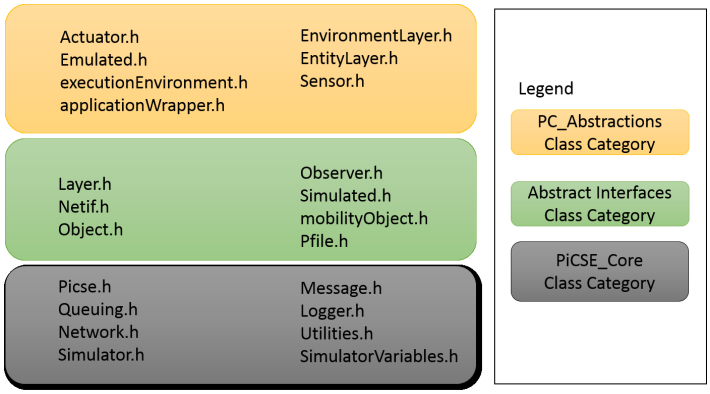

Figure A.2: Class Category distribution of header files.

PC_Abstraction class category

These files are the header files for the class definitions contained with the PC_Abstraction class category. For an explanation of the classes and their architecture, please refer to chapter 3 of this thesis.

environmentalLayer.h

```
#ifndef RPS_LAYER
#define RPS_LAYER

#include "simulator.h"
#include "layer.h"
#include <vector>
#include <list>

///// TO DO
//implement dtors properly and deallocate memory.

/**
 * EnvironmentLayer's are used to model an aspect of the physical
 * environment. Perhaps they would be better refactored to be
 * environmental layers.
 */
```

```
class EnvironmentLayer: public Layer{
 public:
  EnvironmentLayer(long double granularity=100,
    long int period=0);
  /**
   * Abstract populate function called to instantiate values
   * Not sure why this is needed. Clarify!
   */
  virtual void populate() = 0;
  /// Schedule initial events. Again not sure why this is required.
  virtual void setInitialEvents() = 0;
  /**
   * Default is to not subscribe to any events so this is an empty method.
   * Note that this is virtual so inheriting classes can over-write as appropriate.
   */
  virtual void Update(Observable* src){}
  /**
   * Default is to not subscribe to any events so this is an empty method.
   * Note that this is virtual so inheriting classes can over-write as appropriate.
   */
  virtual void Update(Observable* src, int event){}
 protected:
 private:
  void registerSelf();
};



/**
 * Extends the EnvironmentLayer to add a grid of doubles.
 * Also includes double-based set'ers and get'ers
 */
class DOUBLE_EnvironmentLayer: public EnvironmentLayer{
 public:
  DOUBLE_EnvironmentLayer(long double granularity=100,
   long int period=0);
```

```
  ~DOUBLE_EnvironmentLayer(){}
  void setState(Location, double);
  double queryState(Location);
  double getStateAverage();
  double** getGrid();
 protected:
  double** grid;
 private:
};

/**
 * Extends the EnvironmentLayer to add a grid of Int's.
 * Also includes int-based set'ers and get'ers
 */
class INT_EnvironmentLayer: public EnvironmentLayer{
 public:
  INT_EnvironmentLayer(long double granularity=100,
   long int period=0);
  ~INT_EnvironmentLayer(){}
  void setState(Location, int);
  int queryState(Location);
  int getStateAverage();
  //need to make this protected or something
  int** getGrid();
 protected:
  int** grid;
 private:
};

/**
 * Extends the EnvironmentLayer to add a grid of Float's.
 * Also includes float-based set'ers and get'ers
 */
class FLOAT_EnvironmentLayer: public EnvironmentLayer{
 public:
  FLOAT_EnvironmentLayer(long double granularity=100,
```

```
   long int period=0);
  ~FLOAT_EnvironmentLayer(){}
  void setState(Location, float);
  float queryState(Location);
  float getStateAverage();
  float** getGrid();
 protected:
  float** grid;
 private:
};


ostream& operator<<(ostream&,  DOUBLE_EnvironmentLayer&);
ostream& operator<<(ostream&,  INT_EnvironmentLayer&);
ostream& operator<<(ostream&,  FLOAT_EnvironmentLayer&);

#endif
```

entityLayer.h

```
#ifndef REF_LAYER
#define REF_LAYER

#include "simulator.h"
#include "layer.h"
#include <vector>
#include <list>
#include <map>

///// TO DO
// Implement dtors properly, deallocate memory

typedef std::list<Object*> GROUP_OF_OBJECTS;
typedef std::map<string, Object*> FLAT_LIST_OF_OBJECTS;

/// Indicates whether objects update themselves or if layer updates them as a whole.
```

```
enum UPDATE_TYPE {SELF, AUTO_T};
/// Types of events generated
enum ENT_LAYER_EVENT_ID {
  ENT_LAYER_TRANSFORM_EVENT = 0, // Layer has updated
  OBJECT_TRANSFORM_EVENT };      // Object has joined or left

/**
 * A class that contains references to entities.
 * Provides functionality for managing groups of similar types of polymorphic objects.
 * These are grouped into lists of objects which are maintained in a grid.
 * See user documentation for greater detail.
 */
class EntityLayer: public Layer{
 public:
  EntityLayer(
      long double granularity = 100,
      long int period = 0,
      UPDATE_TYPE = SELF); /// default is for objects to update themselves
  ~EntityLayer();
  void addObject(Location,Object*);
  // adds it to a flat list accessed by key string
  void addObject(string, Object*);
  // why not pass in location parameter here?
  void removeObject(Object*);
  // remove object from flat_list
  void removeObject(string);
  // retrieve a pointer to an Object based on string key
  // returns NULL, if no Object found.
  Object* getObject(string);
  /// Updates the grid location of an object.
  void updateObjectsGridLocation(Object* o);
  /// Updates the grid location of an object. This version used after a transfer of ownership has occurred.
  void updateObjectsGridLocation(Object* o, Location previousLocation);
  GROUP_OF_OBJECTS* getObjectsWithinRange(Location, long double);
  GROUP_OF_OBJECTS* getObjectsWithinBoundingRectangle(Location, Location);
  void transform();
```

```
  GROUP_OF_OBJECTS** getGrid();
  FLAT_LIST_OF_OBJECTS* getList();
  /**
   * Default is to not suscribe to incoming events.
   * EntityLayers can be inherited from if a user wishes to subscribe to be notified of published events.
   */
  virtual void Update(Observable* src){}
  virtual void Update(Observable* src, int event){}
  virtual void doEvent(int eid);

  bool closeEnoughToReceive(Location, Object*, long double);

 protected:

 private:
  /// Registers itself with the world as part of the model
  void registerSelf(){World::registerEntityLayer(this);}
  void addUpdatedObject(Location,Object*);
  gridLocation search(Object*);
  GROUP_OF_OBJECTS** grid;
  FLAT_LIST_OF_OBJECTS* flat_list;
  UPDATE_TYPE updateType;
};

ostream& operator<<(ostream&,  EntityLayer&);

#endif
```

sensor.h

```
#ifndef SENSOR
#define SENSOR

#include "environmentLayer.h"
#include "object.h"

enum SENSOR_EVENTS {SENSE=0};
/**
 * A generic class for a sensor which doesn't really do much.
 */
class Sensor: public Object{
 public:
  Sensor(long int t2s,long int p);
  Sensor(long int p=0);
  ~Sensor(){}
  virtual void doEvent(int l);
 private:
 protected:
};


/**
 * A float sensor
 */
class F_Sensor: public Sensor{
 public:
  /// Default sensor is sporadic
  F_Sensor(FLOAT_EnvironmentLayer* src, long int p=0);
  ~F_Sensor(){}
  /// Returns a reading from the source layer using the sensors location
  float getReading();
  virtual void transform();
 private:
  FLOAT_EnvironmentLayer* sensedLayer;
};

/**
 * A double sensor
 */
class D_Sensor: public Sensor{
 public:
  /// Default sensor is sporadic
  D_Sensor(DOUBLE_EnvironmentLayer* src, long int p=0);
  ~D_Sensor(){}
```

actuator.h

```
#ifndef ACTUATOR
#define ACTUATOR

#include "environmentLayer.h"
#include "entityLayer.h"
#include "object.h"

enum ACTUATOR_EVENTS {ACTUATE=0};
/**
 * A generic class for an actuator which doesn't really do much.
 * Provides periodic/sporadic functionality.
 */
class Actuator: public Object{
 public:
  Actuator(long int p=0);
  ~Actuator(){}
  virtual void doEvent(int l);
  virtual void transform(){ actuate(); }
  virtual void actuate() = 0;
 private:
};

/// An object actuator
class Object_Actuator: public Actuator{
 public:
  Object_Actuator(Object* s, long int p = 0);
  ~Object_Actuator(){}
  virtual void actuate(){}
 private:
 protected:
  Object* src;
};

/// An actuator for acting on entity layers
```

```
class EntityLayer_Actuator: public Actuator{
 public:
  EntityLayer_Actuator(EntityLayer* s, long int p = 0);
  ~EntityLayer_Actuator(){}
  virtual void actuate(){}
 private:
 protected:
  EntityLayer* src;
};

/// An actuator for acting on FLOAT based environmental layers
class Float_EnvironmentLayer_Actuator: public Actuator{
 public:
  Float_EnvironmentLayer_Actuator(FLOAT_EnvironmentLayer* s, long int p = 0);
  ~Float_EnvironmentLayer_Actuator(){}
  virtual void actuate(){}
 private:
 protected:
  FLOAT_EnvironmentLayer* src;
};

/// An actuator for acting on DOUBLE based environmental layers
class Double_EnvironmentLayer_Actuator: public Actuator{
 public:
  Double_EnvironmentLayer_Actuator(DOUBLE_EnvironmentLayer* s, long int p = 0);
  ~Double_EnvironmentLayer_Actuator(){}
  virtual void actuate(){}
 private:
 protected:
  DOUBLE_EnvironmentLayer* src;
};


/// An actuator for acting on INT based environmental layers
class Int_EnvironmentLayer_Actuator: public Actuator{
 public:
```

```
  Int_EnvironmentLayer_Actuator(INT_EnvironmentLayer* s, long int p = 0);
  ~Int_EnvironmentLayer_Actuator(){}
  virtual void actuate(){}
 private:
 protected:
  INT_EnvironmentLayer* src;
};

#endif
```

emulated.h

```
#ifndef _EMULATED_IF_
#define _EMULATED_IF_

#include <string>
using namespace std;
#include "execEnv.h"

class Emulated{
 public:
  /// Set the the execution environment
  void setExecutionEnvironment(ExecutionEnvironment* e);
  /// A pointer to the execution environment layer_conor working_backups;
  ExecutionEnvironment* executionEnvironment;
  /// Set attach point for a sensor, or actuator.
  void setAttachPoint(string);
  string getAttachPoint();
  string attachPoint;

};

#endif
```

executionEnvironment.h

```
#ifndef EXECUTION_ENVIRONMENT
#define EXECUTION_ENVIRONMENT

//#include "fcntl.h"
#include <map>
#include <string>
#include "object.h"
#include "netif.h"

using namespace std;

class p_file;
class Emulated;

typedef std::map<std::string, p_file*> dev2pfile;
typedef std::map<int,p_file*> fd2pfile;

class ExecutionEnvironment:public virtual Object, public Net_interface{
 public:
  ExecutionEnvironment();
  void addEmulatedApplication(Emulated* e);
  void addEmulatedHardware(Emulated*,string);

  virtual void transform(){}
  virtual void doEvent(int i ){}
 protected:
  void createFSpace(std::string __file);
  dev2pfile* d2p;
  fd2pfile* f2p;

 private:
  //  int fdcount;
};

#endif
```

applicationWrapper.h

```
#ifndef APPLICATION_WRAPPER_H
#define APPLICATION_WRAPPER_H

#include <pthread.h>
#include "simulated.h"
#include "emulated.h"

#include "observer.h"

enum APP_EVENTS {START_APP=0};
// An program emulation wrapper.

class ApplicationW :public Simulated,public Observable, public Emulated{
 public:
  /// Ctor, schedules application start time, provided execution env is set
  ApplicationW(long int t,ExecutionEnvironment*);
  /// Plain ctor, instantiates variables.
  ApplicationW();
  /// Dtor, needs to be implemented
  ~ApplicationW(){}
  /// Schedules execution t ms into the future
  void scheduleExecution(long int t);
  /// Returns true if execution env is set, false otherwise
  bool isExecEnvSet();
  /// Obtains mutex
  void getLock();
  /// Invokes program. To be implemented by child class
  virtual void run() = 0;
  /// Kills the currentThread after notifying the base thread to restart it.
  void killThisThread();

 private:
  /// Used for thread signalling
  pthread_cond_t thread_cond;
```

```
  /// The thread.
  pthread_t thread;
  /// schedules start event
  void doEvent(int);
  /// Starts the thread
  virtual void exec();
  /// Returns true if program already scheduled to start.
  bool started;
};


// Have to call a c-style func for pthread
// This is a dummy one
extern void* func(void*);


#endif
```

Core_Components Class Category Header Files

These files are the header files for the class definitions contained with the Core_Components class category. For an explanation of the classes and their architecture, please refer to chapter 3 of this thesis.

picse.h

```
#ifndef _PICSE_HEADERS
#define _PICSE_HEADERS

/// The basic headers required.

#include "queuing.h"
#include "world.h"
#include "network.h"
#include "simulator.h"


//////////////////////
// Modify this to include all the basic headers.


#include "object.h"

#include "entityLayer.h"
#include "environmentLayer.h"

#include "objectLogger.h"
#include "environmentLayerLogger.h"
#include "entityLayerLogger.h"

#include "mobilityObj.h"
#include "sensor.h"
```

```
#endif
```

simulator.h

```
#ifndef SIMULATED
#define SIMULATED

/**
 * An interface which all event generating events must inherit from.
 */
class Simulated{
 public:
  /**
   * Abstract method called when an event is performed
   * @param int event id specifies the event to perform
   */
  virtual ~Simulated(){}
  virtual void doEvent(int eventid)=0;
};

#endif
```

world.h

```
#ifndef WORLD
#define WORLD
#include <iostream>


#include <vector>
#include <list>
#include "message.h"

class Net_interface;
class EntityLayer;
class EnvironmentLayer;
```

```
class Object;
class Location;

typedef std::list<EntityLayer*> ENTITY_LAYER_STACK;
typedef std::list<EnvironmentLayer*> ENVIRONMENT_LAYER_STACK;

enum DIMENSIONS{TWO_D =0, THREE_D};


// singleton class.
class World{
 public:
  static World* Instance();
  /// Registers a layer in the world
  static void registerEntityLayer(EntityLayer*);
  static void registerEnvironmentLayer(EnvironmentLayer*);
  /// Returns the origin of the 'world'
  //  static Location getOrigin(){ return *origin;}
  static long double getXSize(){return xSize;}
  static long double getYSize(){return ySize;}
  static long double getZSize(){return zSize;}

  static long double getModelXSize(){return xSize + xOffset;}
  static long double getModelYSize(){return ySize + yOffset;}
  static long double getModelZSize(){return zSize + zOffset;}

  static long double getXOffset(){return xOffset;}
  static long double getYOffset(){return yOffset;}
  static long double getZOffset(){return zOffset;}

  static void setXSize(long double x, long double tX = 0){xSize = x-tX; xOffset = tX;}
  static void setYSize(long double y, long double tY = 0){ySize = y-tY; yOffset = tY;}
  static void setZSize(long double z, long double tZ = 0){zSize = z-tZ; zOffset = tZ;}
  //  static void setOrigin(Location* l){origin = l;}
  /// Gets a random location within the world
  static Location getRandomLocation(DIMENSIONS=TWO_D);
```

```
  static void registerObject(Object*);
  /// Should be in a comms layer, delivers a message to all objects
  //  static void deliverMsg(Location,Msg*);

  ////////////
  //Networking
  static void registerNetIF(Net_interface*);

 protected:
  /// Default ctor
  World();
  ~World();

 private:
  static World* worldInstance;
  /// Location of origin (cartesian system)
  //  static Location* origin;
  /// Scale of the world
  static long double xSize, ySize, zSize, xOffset, yOffset,zOffset,
    TRANSMISSION_RANGE, PROB_DROP_MESSAGE ;
  /// Layer stack
  static ENTITY_LAYER_STACK* EntityLayerStack;
  static ENVIRONMENT_LAYER_STACK* EnvironmentLayerStack;
  //  static void probabilisticDeliver(Msg*, Object*);
};


//ostream& operator<<(ostream&, Location&);



//////////////////////////////////////
class Location{
 public:
  Location(long double xa=0, long double ya=0, long double za=0)
```

```
    : x(xa), y(ya), z(za){}
  long double x; // metres
  long double y;
  long double z;
  void applyOffset(){x -= World::getXOffset(); y -= World::getYOffset();}
  void removeOffset(){x += World::getXOffset(); y += World::getYOffset();}
};


#endif
```

network.h

```
#ifndef NETWORK_SIM
#define NETWORK_SIM

#include <string>
#include <map>
#include "entityLayer.h"

using namespace std;

class Net_interface;
typedef std::map<string, Net_interface*> FLAT_LIST;


/**
   Class representing wireless network. Receivers, represented by Net_interface objects
   can register with the network, at a given location, and are allowed to send and receive
   messages. The sending and receiving of messages is mediated by the Network class. Two factors
   are taken into consideration when delivering messages. The TRANSMISSION_RANGE
   and PROB_DROP_MESSAGE
 */
class Network{
 public:
  static Network* Instance();
```

```
 //////// WIRELESS
 /// Registers a Net_interface object for sending and receiving messages
 static void registerNetIF(Net_interface*);
 /// Broadcast a msg void* to all receivers within distance
 static void broadcastMsg(void*, Net_interface* );
 /// Broadcast a msg string to all receivers within distance
 static void broadcastMsg(string, Net_interface* );
 /// Send a msg void* to a receiver specified by string adr
 static void unicastMsg(void*, string adr, Net_interface* );
 /// Send a msg string to a receiver specified by string adr
 static void unicastMsg(string, string adr, Net_interface* );
 /// Set the transmission range
 static void setTX(long double t){TRANSMISSION_RANGE = t;}
 /// Set the probability of dropping a message
 static void setProbDrop(long double pd){PROB_DROP_MESSAGE  = pd;}


 ///////// WIRED
 static void registerLan_NetIF(Net_interface*);
 /// Send a msg void* to a receiver specified by string adr
 static void unicastLanMsg(void*, string adr, Net_interface* );
 /// Send a msg string to a receiver specified by string adr
 static void unicastLanMsg(string, string adr, Net_interface* );

 ////////
 /// Delivers an event which has been scheduled in a past. This method spawns a thread.
 static void deliverScheduledEvent(NetworkVoidTuple*);
 static void* deliver(void*);
 static void getLock();
protected:
 Network();
 ~Network();

private:
 /// Tracks receivers
 static EntityLayer* wirelessLayer_Channel1;
```

```
  /// Flat list of wireless receivers indexed by address
  static FLAT_LIST* all_wireless_rcvrs;
  /// Flat list of wired receivers indexed by address
  static FLAT_LIST* all_wired_rcvrs;
  /// Maximum Transmission_range. Defaulpt 200m
  static long double TRANSMISSION_RANGE;
  /// Probability of dropping message. Default .05
  static long double PROB_DROP_MESSAGE;
  /// Deliver message void* using PROB_DROP_MESSAGE, delay timeunits into the future
  static void probabilisticDeliver(void*, Net_interface*, int delay = 1);
  /// Deliver message string using PROB_DROP_MESSAGE
  static void probabilisticDeliver(string, Net_interface*, int delay = 1);
  static Network* networkInstance;
};

enum TUPLE_TYPE{STRING=0,VOIDPTR};

// class to represent a network message event
// instances deleted upon returning from delivery callback
class NetworkVoidTuple{
 public:
  // the intended receiver
  Net_interface* rec;
  // pointer message || string message indicated by tupleType.
  void* message;
  string messageS;
  //
  TUPLE_TYPE tupleType;

};


#endif
```

message.h

```
#ifndef ENVIRONMENT
#define ENVIRONMENT

#include <string>

class Msg{
 public:
  Msg(){}
  virtual ~Msg(){}
};

#endif
```

netif.h

```
#ifndef NET_IF
#define NET_IF

#include <string>
#include "object.h"
using namespace std;

/**
 * Extends Object because it has an implicit location
 */

class Net_interface: public virtual Object{
 public:
  /// Default
  //  Net_interface();
  /// Ctor, with address
  Net_interface(/*string a,*/bool=false);
  ~Net_interface();
  /// Send void* parameter to everyone
  virtual void send(void*);
```

```
  /// Send void* parameter to particular address
  virtual void send(void*, string);
  /// Send string to everyone
  virtual void send(string);
  /// Send string to particular address
  virtual void send(string, string);
  /// Abstract deliver API for object delivery
  virtual void deliver(void*)=0;
  /// Abstract deliver API for string delivery
  virtual void deliver(string)=0;
  /// Required for doEvent
  virtual void transform(){}
  /// Returns address if set, "" otherwise
  string getAddress();
  ExecutionEnvironment* getExecutionEnvironment(){return associatedExecEnv;}
 private:
  //  string address;
  ExecutionEnvironment* associatedExecEnv;
 protected:
  void setExecutionEnvironment(ExecutionEnvironment* e){associatedExecEnv = e;}

};

#endif
```

logger.h

```
#ifndef LOGGER
#define LOGGER

#include "observer.h"
#include "simulated.h"
#include "simulator.h"
```

```
enum LOGGER_EVENT { LOG_EVENT = 0};
/**
 * Logger is a class which utilises event subsription services provided in class Logable.
 * Logger is the 'observer' part of observer/observable.
 */
class Logger: public Observer, public Simulated{
 public:
  Logger(){};
  /// Ctor takes a string and opens a file for output
  Logger(string);
  /// Flushs the output stream and then closes it.
  virtual ~Logger();
  /**
   * Sets a period of time within the experiment that is to be logged.
   * @param time2begin is the RELATIVE time in the future that the logging is to begin at.
   * @param duration is the length of time to log for
   */
  void setLoggingWindow(long int t, long int d=Simulator::getDuration()-Simulator::currentTime());
  /**
   * Checks if time for one more event before end of run.
   * To be removed. This is an unnecessary check. Just schedule it, and if run ends, so what.
   * This is more efficient than checking every time
   */
  bool timeForOneMoreEvent();
  /**
   * Set the logging period
   * @param p is the period
   */
  void setLoggingPeriod(long int p);

 protected:
  /**
   * Checks to see if time is within the loggingWindow set in setLoggingWindow()
   * @return true if in window or no window set
   */
```

```
  bool checkValidLogWindow();
  /// Default log event to be implemented by inheritors
  virtual void log() = 0;
  /// Calls default log() method on scheduled log events
  virtual void doEvent(int id);
  /// The output file
  ofstream output;
  /// Time to start logging.
  long int loggingWindowOpening;
  /// Time to stop logging.
  long int loggingWindowClose;
  /// The period of this logger( default is 0)
  long int LOGGING_PERIOD;
  /// Bool indicating whether window set or not
  bool window_set;
};

#endif
```

utilities.h

```
#ifndef UTILITIES_H
#define UTILITIES_H

#include "world.h"
#include <string>
#include <iomanip>
#include <sstream>
#include <cmath>
using std::string;
using std::ostringstream;

class UTILITIES{
 public:
  static long double distanceP2P(Location, Location, DIMENSIONS=TWO_D);
  static long int max(long int, long int);
```

```
  static long int min(long int, long int);
  static string generateUniqueIDFromPrefix(string);
 private:
  static long int idcount;
};

#endif
```

simulatorVariables.h

```
#ifndef SIM_VARIABLES
#define SIM_VARIABLES

const long int SIMULATION_DURATION = 1000*60*60*1; // 12 hours

#endif
```

The Abstract_Interfaces Class Category Header Files

These files are the header files for the class definitions contained with the PC_Abstraction class category. For an explanation of the classes and their architecture, please refer to chapter 4 of this thesis.

layer.h

```
#ifndef LAYER
#define LAYER

#include "world.h"
#include "utilities.h"
#include "simulated.h"
#include "observer.h"

/**
 * Simple class to represent a location in a grid.
```

```
 */
class gridLocation{
 public:
  gridLocation():x(0),y(0){}
  gridLocation(int a, int b):x(a),y(b){}
  int x, y;
};

/// Event's generate by a Layer
enum LAYER_EVENT_ID{LAYER_TRANSFORM_EVENT};

/**
 * A simple abstraction used to model or encapsulate
 * one aspect of the simulation. In class Layer, the notion of the
 * grid is no more than size and granularity. Only in derived
 * layers does this become more concrete.
 */
class Layer: public Simulated,// performs events
     public Observable,  // for logging and also for
     public Observer{
 public:
  Layer(long int period=0, //update period
long double gran = 10, // granularity
long double = World::getXSize(),// default size is that of the world
long double = World::getYSize()//,
// long double = World::getXSize(),// default size is that of the world
// long double = World::getYSize(),
/* Location = World::getOrigin()*/); // default origin of world
  virtual ~Layer(){}
  /// Abstract function to be completed by sub classes
  virtual void transform()=0;
  /**
   * Returns the x size of the grid
   */
  long int getGridXSize(){return gridXSize;}
  /**
```

```
   * Returns the y size of the grid
   */
  long int getGridYSize(){return gridYSize;}
  virtual void doEvent(int id);

 protected:
  /// How often the layer is updated. Can be 0
  long int updatePeriod;
  /// The size of an individual piece of the grid
  long double granularity;
  /// Size of space represented in layer in real terms. mtrs
  long double xSize, ySize;
  /// Dimensions of the grid used in the layer
  long int gridXSize, gridYSize;
  /// The cartesian origin of the grid
  //  Location origin;

 private:

};

#endif
```

netif.h

```
#ifndef NET_IF
#define NET_IF

#include <string>
#include "object.h"
using namespace std;

/**
 * Extends Object because it has an implicit location
 */
```

```
class Net_interface: public virtual Object{
 public:
  /// Default
  //  Net_interface();
  /// Ctor, with address
  Net_interface(/*string a,*/bool=false);
  ~Net_interface();
  /// Send void* parameter to everyone
  virtual void send(void*);
  /// Send void* parameter to particular address
  virtual void send(void*, string);
  /// Send string to everyone
  virtual void send(string);
  /// Send string to particular address
  virtual void send(string, string);
  /// Abstract deliver API for object delivery
  virtual void deliver(void*)=0;
  /// Abstract deliver API for string delivery
  virtual void deliver(string)=0;
  /// Required for doEvent
  virtual void transform(){}
  /// Returns address if set, "" otherwise
  string getAddress();
  ExecutionEnvironment* getExecutionEnvironment(){return associatedExecEnv;}
 private:
  //  string address;
  ExecutionEnvironment* associatedExecEnv;
 protected:
  void setExecutionEnvironment(ExecutionEnvironment* e){associatedExecEnv = e;}

};

#endif
```

object.h

```
#ifndef OBJECT
#define OBJECT

#include <iostream>
#include <string>
#include <list>
#include <map>
#include "world.h"
#include "message.h"
#include "layer.h"
#include "simulated.h"

using std::cout;
using std::endl;
using std::list;
using std::map;

typedef std::map<string,Object*> att;

enum OBJECT_EVENT{MOVEMENT=0};
/**
 * Object class. Objects can be "owned" & "attached". Define what these mean modelwise..
 * Attach probably shouldn't be in here..? These is really part of an emulated abstraction...
 */

class Object:public Simulated, public Observable{
 public:
  ///Default ctor
  Object();
  virtual ~Object();
  ///Preferred ctor
  Object(Location,string = UTILITIES::generateUniqueIDFromPrefix("D_Object"));
  ///A polymorphic function for updating.
  virtual void transform()=0;
```

```
 /// Return string id
 string getID(){ return id; }
 /// Return location
 virtual Location getLocation();
 /// Set location, only valid if owner set to NULL
 void setLocation(Location l);
 /// Return previous location
 virtual Location getPreviousLocation();
 /// Set the previous location
 virtual void setPreviousLocation(Location l){previousLocation = l;}
 /// Set a layer which this object is ref'd in. Can be multiple owners
 void setOwner(EntityLayer*);
 /// Set the updatePeriod, in case you don't want to set it at the constructor
 void setUpdatePeriod( long int i){updatePeriod = i;}
 /// Set the object owner. Used in collocated objects
 void setObjectOwner(Object*);
 /// Add an object to this one. Used in collocated objects scenarios
 void addObject(Object*);
 /// Attach an object to this one physically
 void attachObject(Object*, string);
 /// Update the location of collocated objects
 void updateCollocatedObjects();
 /// Called to instantiateObjs. This should be 'friendly' and not public
 void instantiateObjs();
 /// Return instance of attached device if it exists.
 Object* findDevice(string);
 /// Adds a layer to a list of referers
 void addReferer(EntityLayer*);
 /// Removes a later from the list of referers.
 void removeReferer(EntityLayer*);

protected:
 /// Layer which this object belongs to, what it this used for?
 EntityLayer* owner;
 /// Layers which this object belongs to. These have to be updated as an object moves.
 std::list<EntityLayer*>* layer_references;
```

```
  /// The owner object if this object is collocated
  Object* objectOwner;
  /// String identifier
  string id;
  /// Makes a call to the owning layer to update a grid location
  void updateLayerLocation();
  /// update period of this object
  long int updatePeriod;

 private:
  /// Register object with world
  void registerSelf();
  /// Location of this object.
  Location location;
  /// The previous location of this object
  Location previousLocation;
  /// A container for collocated objects
  std::list<Object*>* collocatedObjects;
  /// A container for attached objects;
  att* attachedObjects;
  bool COLLOCATED_INST;
};

ostream& operator<<(ostream&, Object&);
#endif
```

observer.h

```
#ifndef OBSERVER_H
#define OBSERVER_H

#include <iostream>
#include <vector>
#include <fstream>
#include <string>
```

```
#include "simulator.h"
#include "simulated.h"

using namespace std ;
using std::ofstream;
using std::ios;
using std::string;

enum causality_type{LOOSE=0, STRICT};

class Observable;
////////////////////////////////////////////////////
/**
 * Observer
 */
class Observer{
 public:
  Observer(causality_type=STRICT);
  virtual ~Observer(){}
  /// Called when any event occurs at a logable to which the logger has subsribed to.
  virtual void Update(Observable* ) = 0;
  /// Called when an event, which the logger has subsribed to, occurs at the logable source.
  virtual void Update(Observable*, int)= 0;
  causality_type causalityType;
  int lastEventType;
  int lastEventTime;
  Observable* eventSource;
 private:
};


////////////////////////////////////////////////////

/**
 * Observable is a class which is provides event subscription and notification services.
 * Basically it's the 'observable' part of observer/observable.
```

```
 */
class Observable {
 public:
  Observable();
  ~Observable(){};
  void Attach(Observer*);
  void Attach(Observer*, int);
  void Detach(Observer*);
  void Detach(Observer*, int);
  void Notify();
  void Notify(int);

 private:
  vector<Observer*> generalObservers;
  vector< vector<Observer*>* > eventSpecificObservers;
};

#endif
```

simulated.h

```
#ifndef SIMULATED
#define SIMULATED

/**
 * An interface which all event generating events must inherit from.
 */
class Simulated{
 public:
  /**
   * Abstract method called when an event is performed
   * @param int event id specifies the event to perform
   */
  virtual ~Simulated(){}
  virtual void doEvent(int eventid)=0;
};
```

```
#endif
```

mobilityObject.h

```
#ifndef MOBILITY_OBJECT
#define MOBILITY_OBJECT

#include "object.h"
#include "simulator.h"
#include "observer.h"
#include <string>

const long int MOVER_PERIOD = 1000;
const double MIN_SPEED = 0;  // m/s
const double MAX_SPEED = 10;
const double MAX_PAUSE = 30000;

/* RANDOM_WALK NOT IMPLEMENTED YET*/
enum MOBILITY_MODEL{RANDOM_WAYPOINT=0, RANDOM_WALK, STATIONARY};

enum MOB_EVENT_ID {MOBILITY_EVENT, REACHPOINT_EVENT};

class Ref_Layer;
class Mobility_Object: public virtual Object{
 public:
  ///Default ctor
  Mobility_Object(){}
  ~Mobility_Object(){}
  ///Preferred ctor
  Mobility_Object(Location, MOBILITY_MODEL=RANDOM_WAYPOINT);
  void deliver(Msg*);
  /// Implement your own transform() and doEvent()
  /// if you want to create your own mobility model
  virtual void transform();
  virtual void doEvent(int);
```

```
  void reachWayPt();

 private:
  /// Implements the random waypoint algorithm
  void randomWayPoint();
  void newRandWayPt();
  bool reachWayPtInNextIteration();
  void newRandWayPtParams();
  /// Stationary object movement algorithm
  void stationary();
  /// Random walk movement algorithm
  void randomWalk();
  double currentSpeed;
  double currentAngle;
  Location destination;

  long double lastUpdate;
  long double prevDist2pt;
  long double pauseTime;
  MOBILITY_MODEL mobilityModel;
  void (Mobility_Object::*mobilityFuncPtr)();

};

ostream& operator<<( ostream&, Mobility_Object&);


#endif
```

pfile.h

```
#ifndef PICSE_PFILE
#define PICSE_PFILE

#include <string>
```

```
#include <queue>

class ExecutionEnvironment;

class p_file{
 public:
  p_file();
  ~p_file(){}//implement
  char pullChar();
  void pushChar(char);
  std::string _id;
  char buffer[1024];
  int nextchar;
  int fd;
  pthread_cond_t* activeListener;
  ExecutionEnvironment* activeExecutionEnvironment;
  std::queue<char>* bufferq;
};

#endif
```

A.2 Evaluation Scenarios

STEAM Source Files

The following are the primary header files that are used to define the STEAM evaluation scenario. The main.cpp file defines the setup and initialization of the experiment. The SteamExecEnv.h header file defines the STEAM emulated environment which is the emulated middleware component. TestDevice.h and CommandSensor.h are the definitions of two applications that run on top of instances of that STEAM middleware. Finally, CommandSensorProg.h is an example of an ApplicationWrapper class which is used to initialise the TestDevice and CommandSensor applications.

main.cpp

```
#include <iostream>
#include <sstream>
#include <cstdlib>

#include "network.h"
#include "picse.h"

#include "steamExecEnv.h"
#include "commandSensorProg.h"
#include "soProg.h"
#include "testDeviceProg.h"
#include "mobilityObj.h"

int main( int argc, char **argv )
{

  int simDuration = 1000*60*2.5;
```

```
int iterations = 1;
int worldXSize = 1000, worldYSize = 1000;

// Create a simulation engine
new Future(LINKED); // event list, clock, etc
// Create world
Simulator::Instance();
// Set simulation duration
//Simulator::setDuration(1000*60*2.5);
Simulator::setDuration(simDuration);
// Create a world
World::Instance();
// Set the size of that world
World::setXSize(worldXSize);  World::setYSize(worldYSize);
// Intantiate the network
Network::Instance();

SteamExecutionEnvironment* FIRST_SO[iterations];
SteamExecutionEnvironment* SECOND_SO[iterations];
SteamExecutionEnvironment* MOVING_SENSOR[iterations];
Mobility_Object* mob[iterations];

for(int xk =0 ; xk < iterations ; xk++){

  SECOND_SO[xk] = new SteamExecutionEnvironment(Location(0,0));
  thirdprog* r = new thirdprog();
  SECOND_SO[xk]->addEmulatedApplication(r);
  r->scheduleExecution(1970);
```

```
    // if you want to demo functionality, use fixed locations.
    MOVING_SENSOR[xk] = new SteamExecutionEnvironment(Location(0,0));
    mob[xk] = new Mobility_Object( Location(0,0), RANDOM_WAYPOINT ) ;
    prog* p = new prog();
    MOVING_SENSOR[xk]->addEmulatedApplication(p);
    p->scheduleExecution(1980);
    mob[xk]->addObject(MOVING_SENSOR[xk]);
  }

  // Execute the simulation
  Simulator::run();
  exit(1);
  ////////////////////////////////////////////////////////////////////
}
```

steamExecEnv.h

```
    #ifndef STEAM_EXEC_ENV
    #define STEAM_EXEC_ENV
    #include "execEnv.h"
    #include "rte-impl.h"
    #include "rte-mem.h"
    #include "simulator.h"
    //#include "netif.h"
    class p_file;
    class SteamExecutionEnvironment:public ExecutionEnvironment/*, public Net_interface*/{
     public:
     SteamExecutionEnvironment(Location l);
     ~SteamExecutionEnvironment();
     // RTE - Functions called from RTE_Proximity library
```

```
 int init_rte_publish_local();
 int init_rte_subscribe_local();
 channel_id announce_local(subject sub, proximity* prox, latency lat, period per,ada
 void unannounce_local(channel_id ch);
 int send_event_local(int channel_id, event* event);
 subscription_id subscribe_local(subject sub, receive_event cb);
 void unsubscribe_local(subscription_id ch);
 void register_update_location_cb_local(update_location_cb cb);
 void make_callbacks_for_local(steam_event* st_ev);
 void free_steam_event_local(steam_event* ev);
private:
 // RTE - Functions called from local RTE
 void send_local_event(steam_event* st_ev);
 void* make_announcement(event_channel* our_channel);

 // DUMMY^2-SEAR - Functions called from local RTE
 int init_dummy_sear_subscribe();
 int init_dummy_sear_publish();
 event_channel* reserve_channel(latency lat, float requested_period, proximity* prox
 event_channel* reserve_channel_for_subscribe(subject sub);
 void free_channel(event_channel* handle);
 int send_on_channel(void* msg, event_channel* to);

 // DUMMY-NETIF - Functions called from DUMMY^2-SEAR functions
 void send(void*);
 void deliver(void*);
 // RTE - Variables needed for local RTE
 int publish_initialised;
 int subscribe_initialised;
```

```
    channel_table publish_channels;
    channel_table subscribe_channels;
    int max_channel_id;
    update_location_cb update_location;

    // DUMMY²-SEAR - Variables needed for DUMMY²-SEAR functions
    int send_socket;
    int recv_socket;
    // Necessary to inherit from Net_interface
    void send(string) {}
    void deliver(string) {}
    virtual void transform() {}
};
#endif
```

testDevice.h

```
#ifndef _TESTDEVICE_H_
#define _TESTDEVICE_H_

#include "rte.h"
#include "rte-publish.h"
#include "rte-mem.h"

#include <iostream>
using std::cout;
using std::endl;

#include <string>
using std::string;
```

```
/**
 * @warning  Copyright Trinity College Dublin
 * @class CommandSensor
 * @brief Gives commands to the car.
 * @author Aline Senart
 * @date 05-08-19
 */
class TestDevice
{
 public:
  /** Constructor. */
  TestDevice();

  /** Destructor. */
  virtual ~TestDevice();

};

#endif //_TESTDEVICE_H_
```

commandSensor.h

```
#ifndef _COMMANDSENSOR_H_
#define _COMMANDSENSOR_H_

#include <unistd.h>

#include "rte.h"
#include "rte-publish.h"
```

```
#include "rte-mem.h"

#include <iostream>
using std::cout;
using std::endl;

#include <string>
using std::string;

/**
 * @warning  Copyright Trinity College Dublin
 * @class CommandSensor
 * @brief Gives commands to the car.
 * @author Aline Senart
 * @date 05-08-19
 */
class CommandSensor
{
 public:
  /** Constructor. */
  CommandSensor();

  /** Destructor. */
  virtual ~CommandSensor();

};

#endif //_COMMANDSENSOR_H_
```

commandSensorProg.h (ApplicationWrapper)

```
#ifndef NEW_PROG
#define NEW_PROG

#include "app.h" // from /libs/picse

class prog: public ApplicationW{
 public:
  prog();
  void run();
};

#endif
```

ITS Map Definition File Excerpt

Below is a sample complete Junction Record taken from citycentre.xml, the xml map file that was used to define the road network topology and constraints in the ITS evaluation scenario.

```
    <junctionRec>
        <id>1526</id>
        <type>TL</type>
        <location>
          <xCoordinate>379278.0</xCoordinate>
          <yCoordinate>266013.0</yCoordinate>
        </location>
        <incomingJunction>
          <id>1525</id>
          <numLanes>2</numLanes>
          <linkDistance>141.69333082400175</linkDistance>
```

```
    <maxVelocity>30.0</maxVelocity>
    <outgoingJunctionRef>
      <id>1527</id>
      <actionType>R</actionType>
    </outgoingJunctionRef>
    <outgoingJunctionRef>
      <id>850</id>
      <actionType>S</actionType>
    </outgoingJunctionRef>
  </incomingJunction>
  <incomingJunction>
    <id>1527</id>
    <numLanes>2</numLanes>
    <linkDistance>73.16419889536138</linkDistance>
    <maxVelocity>30.0</maxVelocity>
    <outgoingJunctionRef>
      <id>1525</id>
      <actionType>L</actionType>
    </outgoingJunctionRef>
    <outgoingJunctionRef>
      <id>850</id>
      <actionType>R</actionType>
    </outgoingJunctionRef>
  </incomingJunction>
  <incomingJunction>
    <id>850</id>
    <numLanes>2</numLanes>
    <linkDistance>82.87339742040264</linkDistance>
    <maxVelocity>30.0</maxVelocity>
```

```
      <outgoingJunctionRef>
        <id>1525</id>
        <actionType>S</actionType>
      </outgoingJunctionRef>
      <outgoingJunctionRef>
        <id>1527</id>
        <actionType>L</actionType>
      </outgoingJunctionRef>
    </incomingJunction>
    <outgoingJunction>
      <id>1525</id>
      <numLanes>1</numLanes>
      <linkDistance>141.69333082400175</linkDistance>
      <maxVelocity>30.0</maxVelocity>
    </outgoingJunction>
    <outgoingJunction>
      <id>1527</id>
      <numLanes>1</numLanes>
      <linkDistance>73.16419889536138</linkDistance>
      <maxVelocity>30.0</maxVelocity>
    </outgoingJunction>
    <outgoingJunction>
      <id>850</id>
      <numLanes>2</numLanes>
      <linkDistance>82.87339742040264</linkDistance>
      <maxVelocity>30.0</maxVelocity>
    </outgoingJunction>
  </junctionRec>
  <junctionRec>
```

```
    <id>810</id>
    <type>null</type>
    <location>
      <xCoordinate>379486.0</xCoordinate>
      <yCoordinate>265683.0</yCoordinate>
    </location>
    <incomingJunction>
      <id>801</id>
      <numLanes>1</numLanes>
      <linkDistance>44.384682042344295</linkDistance>
      <maxVelocity>30.0</maxVelocity>
    </incomingJunction>
  </junctionRec>
```

Bibliography

[Benerecetti 01] Massimo Benerecetti, Paolo Bouquet & Matteo Bonifacio. *Distributed Context-Aware Systems.* Human-Computer Interaction, vol. 16, 2001.

[Booch 93] Grady Booch. Object-Oriented Analysis and Design with Applications. Benjamin-Cummings Publishing Co. Inc., 1993.

[Bresciani 04] Paolo Bresciani, Loris Penserini, Paola Busetta & Tsvi Kuflik. *Agent Patterns for Ambient Intelligence.* In Proceedings of 23rd International conference on Conceptual Modelling, November 2004.

[Camp 02] Tracy Camp, Jeff Boleng & Vanessa Davies. *A survey of mobility models for ad hoc network research.* Wireless Communications and Mobile Computing, 2002.

[Campbell 91] Roy H. Campbell, Nayeem Islam, Ralph Johnson, Panos Kougiouris & Peter Madany. *Choices, Frameworks and Refinement.* In 1991 International Workshop on Object Orientation in Operating Systems, pages 9–15, 1991.

[Campbell 08] A.T. Campbell, S.B. Eisenman, N.D. Lane, E. Miluzzo, R.A. Peterson, Hong Lu, Xiao Zheng, M. Musolesi, K. Fodor & Gahng-Seop Ahn. *The Rise of People-Centric Sensing.* Internet Computing, IEEE, vol. 12, no. 4, pages 12 –21, july-aug. 2008.

[Chen 98] F.Z. Chen & X.Z. Wang. *Software Sensor Design Using Bayesian*

Automatic Classification and Back-Propagation Neural Networks. Industrial and Engineering Chemistry Research, vol. 37, no. 10, pages 3985–3991, 1998.

[Dearle 03] Alan Dearle, Graham Kirby & Ron Morrison et al. *Architectural Support for Global Smart Spaces.* In Proceedings of 4th International Conference on Mobile Data Management, 2003.

[Dey 00] Anind K. Dey & Gregory D. Abowd. *The context toolkit: aiding the development of context-enabled applications.* In Workshop on Software Engineering for Wearable and Pervasive Computing, pages 434–441, New York, NY, USA, 2000. ACM Press.

[Esser 97] J. Esser & M. Schreckenberg. *Microscopic Simulation of Urban Traffic Based on Cellular Automata.* Journal of Modern Physics, 1997.

[Eugster 06] Patrick Eugster, Benoit Garbinato & Adrian Holzer. *Pervaho: A Development Test Platform for Mobile Ad hoc Applications.* In Mobile and Ubiquitous Systems: Networking Services, 2006 Third Annual International Conference on, pages 1 –5, july 2006.

[Fall 01] K Fall & K. Varadhan, editeurs. The ns Manual (formerly ns Notes and Documentation). http://www.isi.edu/nsnam/ns/ns-documentation.html (last visited 13 May 2011), 2001.

[Fishwick 95] Paul Fishwick. Simulation Model Design and Execution: Building Digital Worlds. Prentice Hall, 1 edition, 27 January 1995.

[Fleury 07] Pascal Fleury, Jan Curin & Jan Kleindienst. *SitCom - Development Platform for Multimodal Perceptual Services.* In V. Marik, V Vyatkin & A.W. Colombo, editeurs, Proceedings of Internation Conference on Industrial Applications of Holonic and Multi-Agent Systems, pages 104–113. Springer-Verlad Berlin Heidelberg, 2007.

[Fujimoto 00] Richard M. Fujimoto. Parallel and Distributed Simulation Systems. 2000.

[Gamma 95] Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides. Design Patterns:Elements of reusable Object-Oriented Software. Addison-Wesley, 1995.

[Girod 04a] Lewis Girod, Jeremy Elson, Alberto Cerpa, Thanos Stathopoulos, Nithya Ramanathan & Deborah Estrin. *EmStar: a Software Environment for Developing and Deploying Wireless Sensor Networks.* In Proceedings of the 2004 USENIX Technical Conference, 2004.

[Girod 04b] Lewis Girod, Thanos Stathopoulos, Nithya Ramanathan, Jeremy Elson, Deborah Estrin, Eric Osterewil & Tom Schoellhammer. *A System for Simulation, Emulation, and Deployment of Heterogenous Sensor Networks.* In Proceedings of the 2nd ACM Conference on Embedded Networked Sensor Systems (SenSys 04), 2004.

[Guinard 10] Dominique Guinard, Vlad Trifa, Stamatis Karnouskos, Patrik Spiess & Domnic Savio. *Interacting with the SOA-Based Internet of Things: Discovery, Query, Selection, and On-Demand Provisioning of Web Services.* IEEE Trans. Serv. Comput., vol. 3, pages 223–235, July 2010.

[Handte 09] Marcus Handte, Wolfgang Apolinarski, Pedro Marron, Vinny Reynolds, Danh Le Phuoc & Manfred Hauswirth. *PECES Middleware Challenges: On Building the Bridge Between Islands of Integration.* In Paul Cunningham, editeur, eChallenges 2009 Conference Proceedings. Lecture Notes in Computer Science, 2009.

[Handziski 06] Vlado Handziski, Andreas Köpke, Andreas Willig & Adam Wolisz. *TWIST: a scalable and reconfigurable testbed for wireless indoor experiments with sensor networks.* In Proceedings of the 2nd interna-

tional workshop on Multi-hop ad hoc networks: from theory to reality, REALMAN '06, pages 63–70, New York, NY, USA, 2006. ACM.

[He 04] Tian He & Sudha Krishnamurthy et al. *Energy-Efficient Surveillance System Using Wireless Sensor Networks.* In 2nd International Conference on Mobile Systems, Applications and Services (MobiSys '04), June 2004.

[Heckmann 03] Dominik Heckmann. *A Specialised Representation for Ubiquitous Computing and User Modelling.* In Workshop on User Modelling for Ubiquitous Computing at User Modelling 2003 (UM'03), 2003.

[Hill 00] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler & Kristofer Pister. *System architecture directions for networked sensors.* SIGARCH Comput. Archit. News, vol. 28, pages 93–104, November 2000.

[Hynes 09] Gearoid Hynes, Vinny Reynolds & Manfred Hauswirth. *A Context Lifecycle for Web-based Context Management Services.* In Proceedings of the 4th European Conference on Smart Sensing and Context (EuroSSC), 2009.

[Ji 04] Xiang Ji & Hongyuan Zha. *Sensor positioning in Wireless Ad-hoc Sensor Networks Using Multidimensional Scaling.* In Proceedings of IEEE INFOCOM, pages 2652–2661, 2004.

[Jiang 09] Mingxing Jiang, Zhongwen Guo, Feng Hong, Yutao Ma & Hanjiang Luo. *OceanSense: A practical wireless sensor network on the surface of the sea.* In Pervasive Computing and Communications, 2009. PerCom 2009. IEEE International Conference on, pages 1 –5, march 2009.

[John J. Barton 03] Vikram Vijayaraghavan John J. Barton. *UBIWISE, A Simulator for Ubiquitous Computing Systems Design.* Rapport technique, HP

Labs, http://www.hpl.hp.com/techreports/2003/HPL-2003-93.html, 29 April 2003.

[Johnson 88] Ralph E. Johnson & Brian Foote. *Designing Reusable Classes.* JOOP, vol. 1, no. 2, pages 22–35, 1988.

[Jouve 09] W. Jouve, J. Bruneau & C. Consel. *DiaSim: A parameterized simulator for pervasive computing applications.* In Pervasive Computing and Communications, 2009. PerCom 2009. IEEE International Conference on, pages 1 –3, march 2009.

[Klein 01] Lawrence A. Klein. Sensor Technologies and Data Requirements for ITS. Artech House, 2001.

[Kranz 06] Matthias Kranz, Radu Bogdan Rusu & Alexis Maldonado. *A Player/Stage System for Context-Aware Intelligent Environments.* In System Support for Ubiquitous Computing Workshop (UbiSys 2006), 2006.

[Kranz 07] Matthias Kranz, Wolfgang Spiessl & Albrecht Schmidt. *Designing Ubiquitous Computing Systems for Sports Equipment.* In Pervasive Computing and Communications, 2007. PerCom '07. Fifth Annual IEEE International Conference on, pages 79 –86, march 2007.

[Kruchten 95] P. B. Kruchten. *The 4+1 View Model of architecture.* IEEE Software, 1995.

[Kuhl 99] Frederick Kuhl, Richard Weatherly & Judith Dahmann. Creating Computer Simulation Systems: An Introduction to the High Level Architecture. Prentice Hall, 1 edition, 18 October 1999.

[Labella 06] Thomas Halva Labella, Gerhard Fuchs & Falko Dressler. *A Simulation Model for Self-organised Management of Sensor/Actuator Networks.*

http://www7.informatik.uni-erlangen.de/ dressler/publications/fg-selbstorganisation-2006.pdf, 2006.

[Law 00] Averill M. Law & W. David Kelton. Simulation Modelling and Analysis. McGraw-Hill Higher Education, 3rd edition, 2000.

[Levis 03] Philip Levis, Nelson Lee, Matt Welsh & David Culler. *TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications.* In Proceedings of 1st ACM Conference on Embedded Networked Sensor Systems (SenSys 03), 2003.

[Li 06] Ting Li, Freek Hofker & Fred Jansma. *Passenger Travel Behavious Model in Railway Network Simulation.* 2006 Winter Simulation Conference, 2006.

[Lu 09] Hong Lu, Wei Pan, Nicholas D. Lane, Tanzeem Choudhury & Andrew T. Campbell. *SoundSense: scalable sound sensing for people-centric applications on mobile phones.* In Proceedings of the 7th international conference on Mobile systems, applications, and services, MobiSys '09, pages 165–178, New York, NY, USA, 2009. ACM.

[Mangharam 06] Rahul Mangharam, Daniel Weller, Raj Rajkumar, Priyantha Mudalige & Fan Bai. *GrooveNet: A Hybrid Simulator for Vehicle-to-Vehicle Networks.* In Mobile and Ubiquitous Systems: Networking Services, 2006 Third Annual International Conference on, pages 1 –8, july 2006.

[Martin 06a] Miquel Martin & Petteri Nurmi. *A Generic Large Scale Simulator for Ubiquitous Computing.* In Third Annual International Conference on Mobile and Ubiquitous Systems: Networking & Services, 2006 (MobiQuitous 2006), San Jose, California, USA, July 2006. IEEE Computer Society.

[Martin 06b] Miquel Martin & Petteri Nurmi. *A Generic Large Scale Simulator for Ubiquitous Computing.* Third Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services, pages 1–3, July 2006.

[Masson] M. H. Masson, S. Canu & Y. Grandvalet. *Software Sensor Design Based on Empirical Data.* http://www.hds.utc.fr/ em2s/sym/Toulouse/intronew.ps.

[Medagliani 10] P. Medagliani, J. Leguay, V. Gay, M. Lopez-Ramos & G. Ferrari. *Engineering energy-efficient target detection applications in Wireless Sensor Networks.* In Pervasive Computing and Communications (PerCom), 2010 IEEE International Conference on, pages 31 –39, 29 2010-april 2 2010.

[Meier 03] Rene Meier & Vinny Cahill. *Exploiting Proximity in Event-Based Middleware for Collaborative Mobile Applications.* In 4th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS '03), volume 2893, pages 285–296. LNCS, 2003.

[Morla 04] Ricardo Morla & Nigel Davies. *Evaluating a Location-Based Application: A Hybrid Test and Simulation Environment.* In Proceedings of 2nd International Conference on Pervasive Computing, 2004.

[Nakata 07] Junya Nakata, Satoshi Uda, Toshiyuki Miyachi, Kenji Masui, Razvan Beuran, Yasuo Tan, Ken-ichi Chinen & Yoichi Shinoda. *StarBED2: Large-scale, Realistic and Real-time Testbed for Ubiquitous Networks.* In 3rd International Conference on Testbeds and Research Infrastructure for the Development of Networks and Communities (TRIDENTCOM), 2007.

[Narendra 05] Nanjangud C Narendra. *Large Scale Testing of Pervasive Computing Systems Using Multi-Agent Simulation.* In Proceedings of Third IEEE

International Workshop on Intelligent Solutions in Embedded Systems (WISES2005), 2005.

[Naumov 03] Valery Naumov & Thomas Gross. *Simulation of Large Ad Hoc Networks.* In In Proceedings of The Sixth ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM 2003), 2003.

[Nishikawa 06] Hiroshi Nishikawa, Shinya Yamamoto, Morihiko Tamai, Kouji Nishigaki, Tomoya Kitana, Naoki Shibata, Keiichi Yasumoto & Minoru Ito. *UbiREAL: Realistic Smartspace Simulator for Systematic Testing.* In 8th Int'l Conf. on Ubiquitous Computing (UbiComp2006). LNCS4206, September 2006.

[North 06] M.J. North, N.T. Collier & J.R. Vos. *Experiences Creating Three Implementations of the Repast Agent Modeling Toolkit.* ACM Transactions on Modeling and Computer Simulation, vol. 16, no. 1, pages 1–25, January 2006.

[O'Neill 05] Eleanor O'Neill & Martin Klepal. *A Testbed for evaluating Human Interaction with Ubiquitous Computing Environments.* In Proceedings of 1st International Conference on Testbeds and Research Infrastructures for the DEvelopment of NeTworks & COMmunities (TRIDENTCOM), 2005.

[Park 00] Sung Park, Andreas Savvides & Mani B. Srivastava. *A Simulation Framework for Sensor Networks.* In Proceedings of the 3rd ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM 2000), 2000.

[Parker 04] E.L. Parker. *Current Research in Multi-Robot Systems.* Journal of Artificial Life and Robotics, 2004.

[Razafindralambo 10] T. Razafindralambo, N. Mitton, A.C. Viana, M.D. de Amorim & K. Obraczka. *Adaptive deployment for pervasive data gathering in connectivity-challenged environments.* In Pervasive Computing and Communications (PerCom), 2010 IEEE International Conference on, pages 51 –59, 29 2010-april 2 2010.

[Salim 08] F.D. Salim, Licheng Cai, M. Indrawan & Seng Wai Loke. *Road Intersections as Pervasive Computing Environments: Towards a Multi-agent Real-Time Collision Warning System.* In Pervasive Computing and Communications, 2008. PerCom 2008. Sixth Annual IEEE International Conference on, pages 621 –626, march 2008.

[Schmidt 97] Douglas C. Schmidt. *Guest Editorial: Object-Oriented Application Frameworks.* Communications of the ACM, Special Issue on Object-Oriented Application Frameworks, vol. 40, no. 10, 1997.

[Selvarajah 10] K. Selvarajah & N. Speirs. *Integrating Smart Spaces into the PErvasive Computing in Embedded Systems (PECES) Project.* In Consumer Communications and Networking Conference (CCNC), 2010 7th IEEE, pages 1 –2, jan. 2010.

[Senart 06] Aline Senart, Raymond Cunningham, Melanie Bouroche, Neil O'Connor, Vinny Reynolds & Vinny Cahill. *MoCoA:Customisable Middleware for Context-aware Mobile Applications.* 8th International Symposium on Distributed Objects and Applications (DOA 2006), 2006.

[Seo 05] Jinseok Seo, Gwanghoon Goh & Gerard J. Kim. *Creating ubiquitous computing simulators using P-VoT.* In MUM '05: Proceedings of the 4th international conference on Mobile and ubiquitous multimedia, pages 123–126, New York, NY, USA, 2005. ACM.

[Shibuya 04] Kazuhiko Shibuya. *A Framework of Multi-Agent-Based Modeling,*

Simulation, and Computational Assistance in a Ubiquitous Environment. SCS Journal of Simulation, 2004.

[Soanes 05] Catherine Soanes & Angus Stevenson. Oxford English Dictionary. Oxford University Press, 2005.

[Sobeih 05] Ahmed Sobeih, Wei-Peng Chen, Jennifer C. Hou, Lu-Chuan Kung, Ning Li, Hyuk Lim, Hung-Ying Tyan & Honghai Zhang. *J-Sim: A Simulation and Emulation Environment for Wirelesss Sensor Networks.* In Proceedings of the 38th Annual Simulation Symposium (ANSS 2005), 2005.

[Stroustrup 98] Bjarne Stroustrup. *An Overview of the C++ Programming Language*, 1998.

[Sun Microsystems 96] Sun Microsystems. *Wabi 2.2 Users Guide.* Rapport technique, Sun Microsystems, 1996.

[Sundresh 04] Sameer Sundresh, Wooyoung Kim & Gul Agha. *SENS: A Sensor, Environment and Network Simulator.* In Proceedings of IEEE/ACM Annual Simulation Symposium, 2004.

[Tapia 04] Emmanuel Munguia Tapia, Stephen S. Intille & Kent Larson. *Activity Recognition in the Home using Simple and Ubiquitous Sensors.* In Proceedings of 2nd International Conference on Pervasive Computing, 2004.

[Tropper 02] Carl Tropper. *Parallel discrete-event simulation applications.* J. Parallel Distrib. Comput., vol. 62, no. 3, pages 327–335, 2002.

[Verdone 07] Roberto Verdone, Davide Dardari, Gianluca Mazzini & Andrea Conti. Wireless Sensor and Actuator Networks. Academic Press, 21 December 2007.

[Vyas 10] Dhaval Vyas, Anton Nijholt, Dirk Heylen, Alexander Kröner & Gerrit van der Veer. *Remarkable objects: supporting collaboration in a creative environment.* In Proceedings of the 12th ACM international conference on Ubiquitous computing, Ubicomp '10, pages 37–40, New York, NY, USA, 2010. ACM.

[Weiser 91] Mark Weiser. *The Computer for the 21st Century.* Scientific American, vol. 265, no. 3, pages 94–104, September 1991.

[Weiser 93] Mark Weiser. *Some Computer Science Issues in Ubiquitous Computing.* Communications of the ACM, vol. 36, no. 7, pages 74–84, 1993.

[Werner-Allen 05] G. Werner-Allen, P. Swieskowski & M. Welsh. *MoteLab: a wireless sensor network testbed.* In Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on, pages 483 – 488, april 2005.

[Win 07] *Wine: Windows Emulator for x86 Unixes.* http://www.winehq.org/, 2007.

[Xu 10] Xunteng Xu, Lin Gu, Jianping Wang & Guoliang Xing. *Negotiate power and performance in the reality of RFID systems.* In Pervasive Computing and Communications (PerCom), 2010 IEEE International Conference on, pages 88 –97, 29 2010-april 2 2010.

[Yu 05] Liyang Yu, Neng Wang & Xiaoqiao Meng. *Real-time forest fire detection with wireless sensor networks.* In Proceedings of International Conference on Wireless Communications, Networking and Mobile Computing, 2005., volume 2, pages 1214–1217, September 2005.

[Zeigler 00] Bernard P. Zeigler, Herbert Praehofer & Tag Gon Kim. Theory of Modelling and Simulation. Academic Press, 2nd edition, 2000.

[Zomaya 96] Albert Y. H. Zomaya. Parallel and Distributed Computing Handbook. McGraw-Hill Professional, 1996.