A Common Model for Ubiquitous Computing

by

Michael Anthony Blackstock

B.A.Sc., The University of British Columbia, 1991
M.Sc., Simon Fraser University, 2002

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

The Faculty of Graduate Studies

(Computer Science)

The University Of British Columbia

October, 2008

Abstract

Ubiquitous computing (ubicomp) is a compelling vision for how people will interact with multiple computer systems in the course of their daily lives. To date, practitioners have created a variety of infrastructures, middleware and toolkits to provide the flexibility, ease of programming and the necessary coordination of distributed software and hardware components in physical spaces.

However, to-date no one approach has been adopted as a default or de-facto standard. Consequently the field risks losing momentum as fragmentation occurs. In particular, the goal of ubiquitous deployments may stall as groups deploy and trial incompatible point solutions in specific locations. In their defense, researchers in the field argue that it is too early to standardize and that room is needed to explore specialized domain-specific solutions.

In the absence of an agreed upon set of standards, we argue that the community must consider a methodology that allows systems to evolve and specialize, while at the same time allowing the development of portable applications and integrated deployments that work between between sites.

To address this we studied the programming models of many commercial and research ubicomp systems. Through this survey we gained an understanding of the shared abstractions required in a core programming model suitable for both application portability and systems integration.

Based on this study we designed an extensible core model called the Ubicomp

Common Model (UCM) to describe a representative sample of ubiquitous systems to date. The UCM is instantiated in a flexible and extensible platform called the Ubicomp Integration Framework (UIF) to adapt ubicomp systems to this model.

Through application development and integration experience with a composite campus environment, we provide strong evidence that this model is adequate for application development and that the complexity of developing adapters to several representative systems is not onerous. The performance overhead introduced by introducing the centralized UIF between applications and an integrated system is reasonable. Through careful analysis and the use of well understood approaches to integration, this thesis demonstrates the value of our methodology that *directly* leverages the significant contributions of past research in our quest for ubicomp application and systems interoperability.

Table of Contents

List of Tables

List of Figures

List of Programs

Glossary

Adapter Software component that maps heterogeneous data, interfaces and protocols to a common model and data format. 83

AdapterManager The UIF *Message Broker* implementation that mediates interaction messages between the UIF and its adapters. 99

Component Container UIF subsystem that hosts “native” UCM components instantiated by the system. 89

Context Widget A Context Toolkit software component that provides access to context information in their operating environment. Applications can query their state or subscribe to context changes[34]. 29

DataObject UIF internal object used as a generic data structure designed for marshaling to SOAP and integrated systems. 93

Dataspace A data sharing service similar to a tuplespace used in the EQUIP Component Toolkit (ECT) to relay events and share state between software components [46]. 25

Discoverer A Context Toolkit software component used by applications to locate components such as Context Widgets and Aggregators that are of interest to them based on the attributes (e.g., location, username) it is interested in.[34]. 29

Enterprise JavaBeans A component model for component transaction monitors. There are three types of server-side components called enterprise beans: entity, session and message-driven beans [1]. 115

Entity Aggregator A Context Toolkit software component that is acts as a mediator between applications and Context Widgets. It is responsible for all of the context about a particular entity (person, place or thing).[34]. 29

Environment Composition Logic (ECL) UIF subsystem that dispatches calls to integrated ubicomp systems and maintains event subscriptions for applications. 89

Event Heap A tuplespace based coordination system used in iROS where tuples are called *events*, and contain certain mandatory fields for sequencing and garbage collection [64]. 33

Façade In the Façade design pattern [44], the *façade* is an object that provides a simplified interface to a larger body of code. In the Ubicomp Integration Framework, the Façade is a Java object exposed using Web Services that provides a single interface to an integrated ubicomp environment. 81

ICrafter A service infrastructure for the iROS system that includes service aggregation and user interface creation and selection [82]. 53

Java 2 Enterprise Edition Version of the Java Platform used for the development and deployment of enterprise applications [97]. The latest version of this system is now simply called Java Platform, Enterprise Edition (Java EE) [98]. 92

[1] [75] pp 23-24

JavaBeans A software technology for building reusable Java components called "beans". Beans are Java classes that follow a convention for naming, construction and behavior for reuse and manipulation visually in a builder tool [96]. 25

Message Broker A software intermediary that broker's messages between integrated systems. 6

Model and Reasoner UIF subsystem that maintains the current environment model including the UCM itself, specializations of the UCM, entity instances, static context values, capabilities, component descriptions and their relationships using a knowledge base and associated reasoning engine. 89

OWL Web Ontology Language. 56

RDF Resource Description Framework. 10

Session Bean A type of server-side component used in Java-based component transaction monitors typically used to implement application logic. See also Enterprise Javabean [2]. 128

[2] [75]

Acknowledgements

This work would not have been possible without the help, encouragement and financial support of my supervisors Dr. Rodger Lea and Dr. Charles 'Buck' Krasic. It is difficult for me to overstate my appreciation to Dr. Lea. After first meeting he quickly become not only a good friend but a mentor, helping me focus my thesis research, providing valuable feedback and by introducing me to other researchers in the ubiquitous and pervasive systems community.

I wish to thank Dr. Krasic for taking me on as his first PhD student. I am grateful for his open door policy, for the time that he spent with me brainstorming, challenging my ideas, for his feedback and support both academically and career-wise.

Thank you to the National Sciences and Engineering Research Council of Canada for their finanical support for my first two years of study, and for the University of British Columbia University Graduate Fellowship program for my third year.

I'd also like to thank all of my friends and colleagues. While I can't thank everyone, I must single out a few people: Kan Cai, Matt Finke, Tony Tang, Aiman Erbad, Nicole Arksey, Phillip Jeffrey, Meghan Deutcher, Rock Leung, Nels Anderson, Vincent Tsao, Crystal Giesbrecht, and Gavin Tian for their collaboration and feedback. Thank you to Dr. Adrian Friday for his help organizing Ubisys and CMPPC workshops, and for his feedback during his visit to MAGIC and for his

hospitality during my visits to Lancaster.

I cannot finish without saying how grateful I am to my family for their encouragement and support. I dedicate this thesis to my beautiful wife Kim, my sons Danny and Marcus, and to my parents.

Chapter 1

Introduction and Motivation

Ubiquitous computing (ubicomp) is a compelling vision for how computing resources will become an integral part of our daily lives [108]. In future living and working environments, such as our homes [25], schools [49], meeting rooms [83] and hospitals [16], sensors and services embedded in an environment can be used by applications hosted on portable devices such as laptop computers, smart phones, personal entertainment devices, or in the environment itself.

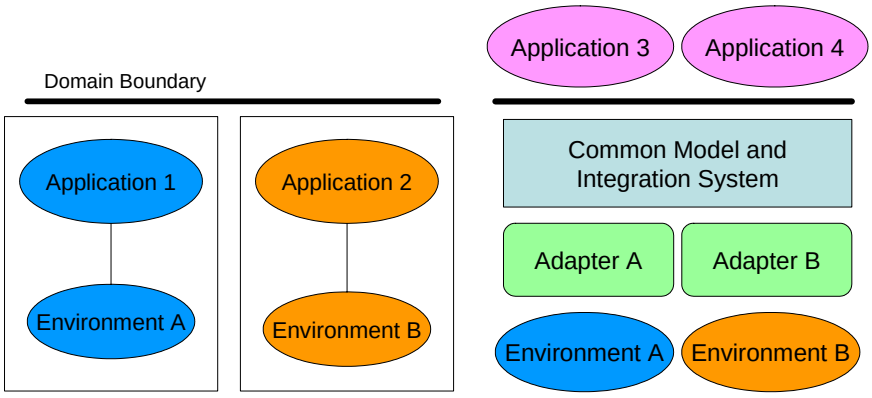
To support this vision, a variety of middleware, toolkits, and operating systems have been created. Over time, these innovative and pioneering systems have matured to address many research issues both unique to ubiquitous computing and/or inherited from distributed systems. These include such issues as hiding the heterogeneity of underlying infrastructures, scalability, dependability, security, privacy, spontaneous interoperation, mobility, context awareness, context management, application mobility, and human computer interface issues [32]. While this compelling vision has inspired much research in all of these areas, after more than 15 years ubicomp systems have (mostly) been confined to lab prototypes and relatively limited deployments. Many reasons have been cited for this lack of deployment [33]. In some cases it can be difficult to persuade others to use a non-standard technology; perhaps there is a perception that, as research prototypes, software quality and ongoing support from their creators will be questionable [95]. That said, even when researchers have attempted to leverage open infrastructure and

middleware standards such as the Open Systems Gateway Initiative (OSGi) [80] or Web protocols (e.g. [68]) ubicomp systems still have not been widely adopted.

Without agreed apon standards, or even a set of best practices on how to build ubicomp systems, designers have primarily focused on supporting applications and user access within single administrative or network domains. This has led to 'islands' of ubicomp installations specialized for specific applications, physical locations and devices. Unless a solution can be found, the lack of standards or a common platform will continue to be an impediment to the widespread adoption of ubicomp systems in the 'real world'.

To bridge islands of existing and future ubicomp deployments, we propose the development of a *common model* for ubicomp systems that facilitates a mapping to existing ubicomp systems' programming models. This will allow developers to create new applications, assured that a suitable intermediary or gateway will allow their application to run on existing systems. It will also support the development of cross domain applications, allowing developers to bridge islands of ubicomp deployments. Furthermore, we argue that the development of a common model is a first step toward standardization of ubicomp deployments and the establishment of a common reference platform.

The current deployment situation for many ubicomp systems is illustrated in Figure 1.1 (a). Application 1 and 2 are dependent on ubicomp environments A and B respectively. These applications can only access environment resources that they are designed for, and only (typically) within the same administrative and network domain. With a common model and supporting infrastructure, applications can make use of environment resources across domain boundaries, independent of location and ubicomp system used as shown in Figure 1.1 (b). Moreover, disparate ubicomp systems can be integrated under a single integrated environment model as shown. Environments can be composed to integrate environment resources such



(a) Existing ubicomp deployments (b) Integrated deployments

Figure 1.1: Current and future integrated ubicomp deployments under a common model.

as smart meeting rooms, and classrooms across larger physical locations such as a cities, campuses or buildings or across branch locations of a company.

The question then arises, if a common model for ubicomp is needed for application portability and systems interoperability, why one has not been proposed by the research community? To understand these issues, the author helped organize two workshops: Systems Support for Ubiquitous Computing (Ubisys) at Ubicomp 2006, and Common Models and Patterns for Pervasive Computing (CMPPC 2007) at Pervasive 2007.

Several submissions and discussions at Ubisys '06 were focused on common abstractions, and models toward interoperability and portability supported by systems and tools[70]. At CMPPC 2007, participants agreed that many *existing* systems can serve as examples for techniques, best practices and design patterns for ubicomp[1]. Both of these workshops helped shape this thesis, especially the dis-

[1] see `http://www.ubisys.org/index.php/Main/CmppcResults`

cussions on patterns, techniques and guidelines at CMPPC[2]. We have begun to document the collective experience of the ubicomp and pervasive systems community into an online resource[3]. In one area of this community portal we have documented specific patterns and techniques such as the use of *Event Brokers*, *Web services*, and *entity aggregation* used in a variety of systems and captured in our work [20–23, 40].

While consensus on a programming model is recognized an important long term goal, this workshop participation highlighted the fact that agreement on one is far from straightforward for two reasons. One is that the field continues to evolve. Researchers are still exploring not only implementation issues, but the abstractions and programming models for ubicomp applications themselves (e.g. [14, 48, 56]). While creating a reference platform is eventually achievable, we feel it is impractical to settle on one while new abstractions and implementations continue to be explored. Furthermore, we argue that even in the long term, the use of the same programming model and supporting middleware platform in all smart spaces is not realistic as there will always be cases where systems specialized for specific applications and locations will be deployed in particular environments.

Given these observations, it is clear that there will *always* be a variety of ubicomp platforms and hence, there is a need to develop a methodology that supports the integration of specialized platforms as they evolve. This will allow developers to continue to create environment-specific applications, while others can create integrated ubicomp deployments, and applications that are portable between sites and administrative domains. Such an approach will ensure systems developers can continue to evolve their platforms while supporting a growing application developer community. However, the development of such a methodology is not straightfor-

[2] see `http://www.ubisys.org/index.php/Main/PatternsTechniquesGuidelines`
[3] `http://www.ubisys.org`

ward, and there are several technical and non-technical challenges to address.

1.1 Challenges

Unlike systems deployed within a single domain, services used between organizations over the wide area Internet are exposed and managed carefully by administrators. Maintaining autonomy and ensuring underlying services are accessed in a secure manner by authorized users is therefore critical.

It is also important to provide a mechanism for resolving protocol and interface mismatches between systems. In closed environments, this is easily resolved by using the same types of hardware, or by wrapping devices and software services using the same middleware. Previous efforts to address this have included the use of device-hosted middleware that expose a single interface for service discovery and binding mapped to various service architectures [43, 45] or middleware that advocates a generic set of interfaces for the rapid assembly of components [47, 76]. While these approaches suffice when integrating individual services and components, they are not feasible when entire middleware systems with a variety of APIs and programming models must be integrated.

Another problem is that ubicomp systems to date support a range of abstraction levels. Some systems expose *high level* abstractions such as explicit *environment models* (e.g. [11, 25]) while others support simpler component or service abstractions (e.g. [47]). Systems have been designed to support a range of deployments from small scale, ad hoc, single task-oriented configurations of devices [76], to large scale integrated campus environments [49]. A system that integrates these systems must be able to "understand" not only a simple component or service abstraction, but higher level concepts exposed by existing ubicomp APIs such as physical *entities* (people, places, things), *context* [35] and *environments*.

Since ubicomp environments are highly dynamic, any supporting system must be self organizing to some degree. An integration platform must be able to adapt to the addition and removal of resources exposed by a variety of internal middleware platforms.

Finally, users should be able to access computing resources around them using a variety of personal computing devices such as PDAs, personal music players or mobile phones using a wide variety of client software such as mobile browsers and custom clients.

1.2 Integration Approach

To begin to address these challenges, we can look toward progress in other domains such as enterprise application integration (EAI). The development of distributed enterprise applications has been supported by the rapid evolution of middleware technology. This technology has moved from supporting single-tier mainframe applications, to distributed object technologies such as Common Object Request Broker Architecture (CORBA) [79], and eventually to the use of message brokers to integrate complete heterogeneous applications across an organization.

Similarly, designers of ubicomp environment systems have leveraged traditional middleware technologies to address issues such as device and service interface heterogeneity. Like many ubicomp systems, enterprise systems have also been restricted for use in single local area networks for various reasons. While architectures based on distributed objects and message brokers have proved effective in integrating applications in a single network, they have not been effective *between* enterprises and network domains. One of the biggest problems in achieving inter-enterprise integration has been the lack of standards at the middleware and component levels. To address this, the enterprise integration community has

turned toward the use of Web technologies.

The Web emerged initially as a technology for sharing information across the Internet. With the introduction of Web services, however, it has also become a medium for application integration. More specifically, Web services have been established as a way to expose the functionality of an information system (or group of systems) to applications in other companies, across network and administrative domains. Web service standards have already made significant progress toward resolving the limitations of conventional middleware platforms. Standardization in key areas needed for cross-domain interoperation have included an interface definition language [105], wide-area service discovery [2] and protocols for tunneling procedure calls within HTTP requests and responses [104]. Web services address a specific purpose: to expose functionality in an information system and make it discoverable and accessible over the wide area Internet in a controlled manner. Conceptually Web services are *wrappers* to encapsulate one or more applications with a unique interface available across the Internet.

When we compare the evolution of ubiquitous computing environments with that of enterprise application integration, they have followed a similar trend. Initially, small scale ubicomp research projects assumed homogeneous interfaces to sensors and actuators, and processing related to the environment and the application has been blurred. To minimize the amount of programming required to create new applications, researchers have recognized the need to support heterogeneous device interfaces, and have provided middleware services to isolate application-specific logic from issues related to device heterogeneity, location, protocols, and non-application specific processing of sensor (context) information.

More recently, service-oriented architectures such as Universal Plug and Play [74] and OSGi [80] have been leveraged for even greater modularity and extensibility within an environment [51]. While the use of standard protocols and service

oriented architectures such as those supported by Web services are necessary first steps toward cross domain interaction, they are not sufficient for application portability and ubicomp system interoperability. Now that internal middleware for ubicomp environments has matured for single domains, deployment of interoperable systems that reach across domains will require appropriate *external* middleware and a common programming model for ubicomp. To achieve this, we must also consider the higher level abstractions and programming models exposed by ubicomp systems so far.

1.3 Ubicomp Programming Models

Many ubicomp systems to date have focused on providing an easily understood programming model to access sensors, services and other environment resources (e.g. [11, 36]). Others have focused on service or device interoperability (e.g. [47, 74, 76]) or large scale infrastructures for sharing context information [58, 77]. Designers have focused on addressing the requirements specific application domains such as collaboration, or locations such as meeting rooms and the home. We maintain that there is such diversity in the deployment objectives and approach of an individual system that no one system is suitable for both application portability and integration of other systems.

One issue is that a given system does not consider the variety of programming models that *other* systems expose for effective interoperability and integration. Rather, they aim to provide a homogeneous interface to the variety of services, sensors and actuators in a single environment. If the chosen integration platform's programming model provides low level abstractions, it may not *leverage* many of the higher level capabilities available to application developers by an underlying system. Conversely, if the integrated programming model is too high level, it may

not be able to *compensate* for the missing abstractions in an integrated system.

For example, the ParcTab system maintained a set of variables aggregated by *environment servers* representing entities such as people, places or groups [91]. Similarly, the Context Toolkit used *Entity Aggregators* to provide a 'one stop shop' for context data about an entity [36]. These systems both made it easier for applications to find relevant context by aggregating information around entity components. Both iROS [83] and Gaia [87] systems highlight the value of a multi-device publish-subscribe *event* infrastructure for smart spaces. Applications can listen for, and produce events to interact with multiple devices. Both of these systems provide well understood and useful abstractions for application development, however, we maintain that it is not straightforward to use the ContextToolkit to expose the capabilities of the EventHeap that iROS applications expect – there is no central event producer in the system. Nor is it an easy task for the iROS system to provide a way for applications to find and query an entity aggregator component – none is defined explicitly in the iROS programming model. A new system designed for abstraction mapping is required. The abstractions this system supports must be based on a thorough analysis of these important systems and others.

1.4 Research Focus

While there are many challenges in creating a common model that lends itself to application portability and interoperability between systems, our research focuses on the following:

- The design of an extensible core model for adequately describing a representative subset of existing ubiquitous computing environments deployed to date. This model must lend itself to application portability and interoperabil-

ity between different environment domains such as the home, the office and public places.

- Providing a flexible and extensible platform to adapt representative systems to this model. This includes systems that support small to large scale deployments, those that expose a range of programming abstractions not specific to ubicomp like services and components to more ubicomp-specific abstractions like context and entities.

The foundation of this work is the design of the Ubicomp Common Model (UCM), a programming model that aims to unify the abstractions of a variety of existing ubicomp systems. We describe the UCM using Semantic Web languages: the Web Ontology Language (OWL) [102] built on the Resource Description Framework (RDF) [101]. With such a model, we hypothesized that an integration system can be designed to map a single API to the interfaces of existing systems with adequate coverage of the underlying functionality. The design of the UCM is based on the following assumptions:

1. The programming models of ubicomp systems deployed to date share certain programming abstractions specific to the ubicomp domain.

2. These abstractions can form the basis for a core programming model suitable for the development of *interoperable applications* that can make use of some subset of the functionality of any underlying ubicomp system.

3. This core programming model can be used to unify the programming models and capabilities of more than one system into *composite* environments.

To validate these claims, we conducted an extensive survey and analysis of existing commercial and research ubicomp systems. Based on this analysis we

identified the abstractions that recur in the programming models of several systems: *environment models*, *entities*, *context*, *entity relationships*, *services*, *events* and *data*. These abstractions formed the basis for the design of the UCM, a core model for ubicomp systems.

To evaluate the UCM, we developed an integration platform called the Ubicomp Integration Framework (UIF). The UIF is a flexible and extensible *meta*-middleware platform based on Web services standards used to integrate ubicomp systems using the UCM and expose its capabilities to applications across network and administrative domains. With this system we can provide an API for application interoperability and portability while allowing underlying systems to continue to specialize and evolve.

This thesis provides details of our analysis and design of the UCM, the design and implementation of the UIF, and evaluation of both for application development and integration with several representative systems. The thesis for this dissertation is as follows:

> *The identification of common abstractions used by existing ubicomp systems contributes to a core common model for integrated ubiquitous computing environments suitable for* both *application interoperability and mapping to existing ubicomp systems' interfaces.*

The primary contributions of this thesis are:

1. A comprehensive survey of existing systems categorizing systems in terms of their *level of abstraction* and *scale of deployment* highlighting the common abstractions used by these systems exploited in the design of a common model for ubiquitous computing.

2. The design of a core model for ubiquitous computing called the Ubicomp

Common Model shown to unify the exposed abstractions of several representative ubicomp systems.

3. A demonstration of the feasibility of using the UCM to map to the abstractions of several representative systems.

4. Confirmation that this model is adequate for application development.

Our secondary contributions include:

1. A novel meta-middleware architecture and implementation for integrating more than one ubicomp system under a common model.

2. The use of an integrated knowledge base and reasoning to describe and maintain an integrated ubicomp environment composed of more than one ubicomp system.

1.5 Thesis Outline

The remainder of this thesis is organized as follows. In Chapter 2 we present a comprehensive survey and analysis of representative ubiquitous systems. Based on this survey we identify the common programming abstractions used as the basis for the UCM design. In Chapter 3 we present the design of the Ubicomp Common Model including the three key aspects of the model: the Environment State, Meta-state and Implementation. In Chapter 4 we describe the design and implementation of the Ubicomp Integration Framework, used to evaluate the UCM by providing a single environment model and associated API to four underlying systems [36, 40, 47, 83]. Our evaluation and discussion is presented in Chapter 5 where we discuss the development of several prototype applications, integration

adapters and performance of the UIF. We conclude the dissertation in Chapter 6 with a discussion of lessons learned and future work.

Chapter 2

Survey and Analysis of Ubiquitous Systems

The design of a common model for ubicomp suitable for integration depends on an understanding of existing systems' programming abstractions. As in any software engineering task, the use of abstraction in a ubicomp system has two main benefits. Firstly, it helps manage complexity for developers by exposing important aspects of a system hiding unimportant details. Secondly, it separates aspects of a system that are common to all from those specific to a particular implementation. In this chapter we survey a broad range of representative ubicomp systems to highlight the individual programming abstractions they expose to application developers. Based on this survey, we then derive a set of abstractions that occur across several systems, giving names and examples for each. These abstractions are the basis of the Ubicomp Common Model presented in Chapter 3.

When we survey the wide variety of systems, we note that finding the most appropriate programming model is not straightforward: there are often tradeoffs related to finding the "right" level of abstraction. For ease of application development, "high level" abstractions can relieve the developer from having to deal with certain implementation details, but for broad applicability it is often more feasible to expose "low level" abstractions that expose more details to the developer.

For example, some ubicomp systems expose distributed components or ser-

vices to applications, what we consider "low level" abstractions. Developers create applications by composing these building blocks by various means. This includes the use of generic interfaces and mobile code [76], linking compatible component properties [47], or through the use of standard interface definitions [74]. Other systems provide higher level abstractions that more closely reflect the overlap of physical and digital space in an effort to make programming pervasive spaces more intuitive. To do this, systems will often associate relevant information and services with physical or virtual *entities* such as people, places, and things. These entity abstractions often act as aggregators or containers for relevant information from sensors and inference services called *context*. Context may include the user's current location and activity, the sound and lighting levels in a room, the online status of a printer, or the names of people in a group for example.

The most appropriate programming model may also depend on the *scale* of a typical deployment. In smaller scale deployments, where the aim is to support simple tasks, a programming model consisting of compositions of service abstractions is often suitable. In room to building-scale deployments with more concurrent users and applications, programmers can benefit from the use of higher level abstractions such as explicit *environment models* that hide individual component implementations. In even larger scale deployments such as a university campus, exposing every projector, light switch, media player, large screen display, camera, and thermostat does not scale for user interfaces, applications or integrators. Furthermore, wide area communications can be slow and expensive. Consequently, wide area systems tend to aggregate functionality around coarser grained entities, even whole environments federated using wide area protocols.

In this chapter we survey and analyse existing systems to identify and name the abstractions that occur regularly in their programming models to derive the core abstractions for a common model for ubicomp. The remainder of this chapter

is organized as follows. In Section 2.1 we define our categorization of ubicomp system. In Sections 2.2 to 2.5 we describe each system by category. In Section 2.6 we discuss the programming abstractions typically exposed in each category and then group these to present our derived set of common abstractions across categories. We conclude this chapter in Section 2.7.

2.1 Categories of Ubicomp Systems

Given the diverse research targets and approaches to systems design, grouping the broad range of ubicomp systems deployed so far can be challenging. For the purpose of this survey, we have gathered systems into four categories as follows:

- **Component Composition Systems**: generally lower level abstractions and smaller scale deployments
- **Context Frameworks**: medium level abstractions/mid scale environments
- **Smart Space Systems**: higher level abstractions/mid scale environments
- **Wide Area Systems**: low to high level abstractions/large scale deployments or cross domain access

These categories were derived by rating systems in terms of two dimensions: the *level of abstraction* a system exposes and the *scale* of a typical deployment, both defined next.

2.1.1 Abstraction levels

To create these four categories we first rated systems in terms of *level of abstraction* to group systems with similar programming models. We point out that our use of

the phrase "level of abstraction" does not indicate the quality or applicability of one system over another, but rather its intended purpose and similarity between systems' programming models. Our definition of abstraction levels including some examples are defined as follows .

- **Service and Component Compositions (low)**. Systems that provide a service oriented architecture or component abstraction are considered to expose "low level" abstractions. Some of these systems will support the composition of components and services. Example systems include several commercial standards like UPnP [74] and research systems such as SpeakEasy/Obje [76].

- **Entities and Context (medium)**. We consider systems that provide abstractions including the notion of a person, place or thing (i.e. physical or virtual entities), and context to expose "medium level" abstractions. Systems at this level often build on distributed services and component architectures. Examples include the Context Toolkit [36] and the Java Context Aware Framework (JCAF) [15].

- **Explicit Detailed Environment Models (high)**. For our survey we rate systems as having a "high level" of abstraction when they provide an interface to an explicit model of the physical environment. The system may include centralized servers for event brokering and data storage. The environment model may include both mobile and fixed entities like places, tables, walls and relationships. Example projects include Sentient Computing [54], the Nexus project [57] and ontology based systems (e.g. [28, 51]). With an explicit and centralized model it is possible for the supporting system to reason about the situation as a whole, removing the need for applications to maintain their own model of the current physical and run time environment.

2.1.2 Scale of Deployment

To categorize systems we also consider the *scale* of deployment targeted by the designers. Some systems, for example, focus on the composition of devices for a single mobile user, or only a few users. A given component configuration will generally support only only a few simple tasks at a time. Other systems aim to integrate several applications, services, devices for all of the users in a room. Medium scale systems tend to support tens of users, in a meeting room for example, and many applications at once. Larger scale systems, designed for wide area deployments and infrastructures with many users and applications, target campuses, cities or the world. To summarize:

- **Small scale**. Sub room, single or few users e.g. on a broadcast network for one (mobile) or a small number of users. This includes ADS, PCOM and the ZeroConfig standards for example.
- **Mid scale**. Room, building, floor, single servers, tens of users, several applications. Systems include those that use servers such as iROS and the Context Toolkit.
- **Large scale**. Campus, city, global, wide area networks, federated servers, wide area protocols for many applications and thousands to millions of users. Systems here include Cooltown and ActiveCampus.

When we place systems along both the abstraction level and scale axes, as shown in Figure 2.1 we note that there is some correlation between these two dimensions. Smaller scale systems tend to use service composition (low level) abstractions since they are often concerned with composing individual components corresponding to devices or software services. When the number of devices, users, and applications increases in medium scale (room-building) systems we see a trend

to provide context and entity (medium level) abstractions that aggregate software services and shared state around objects of interest such as people, places and things - medium level abstractions. Finally, larger systems scale to even more users and applications; the abstractions exposed in larger scale systems tend to range from medium to high level explicit environments.

Interestingly, while there is some correlation between scale and abstraction level, there are examples in the Smart Space and Wide Area Systems categorise that show these dimensions are orthogonal: medium scale systems with high level abstractions (e.g. EasyLiving [25]) and large scale systems that provide medium level abstractions (e.g. Cooltown [68]). Based on their approximate location in the scale/abstraction space, we grouped systems into the four system categories used in the body of this survey as shown.

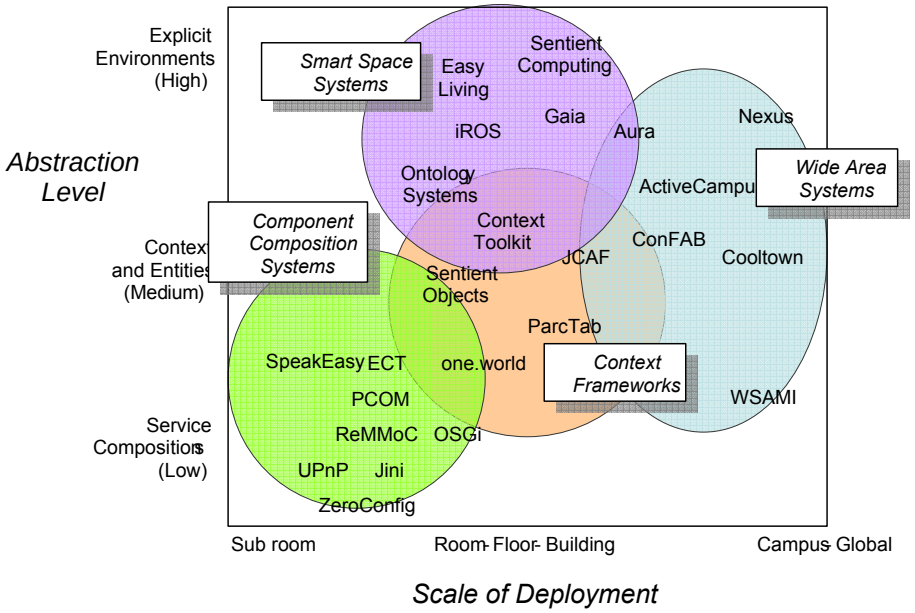


Figure 2.1: Ubicomp systems surveyed placed in terms of scale and abstraction level.

2.1.3 Organization

In the following four sections we present representative systems in each category: Component Composition Systems, Context Frameworks, Smart Space Systems, and Wide Area Systems in rough chronological order. This not only gives a sense of the features of a category, but also of the evolution of abstractions within that category as research progressed. Where a system can be considered to be in more than one category we have placed it in the section where it shared the most features of other systems. In each section we begin by providing a table with a row for each system, summarizing the purpose of its design (e.g. programmability or interoperability or both), its scale, abstractions, and key references. This is followed by a description of the system with attention to the programming model and abstractions it exposes. To respect the significant contributions of these projects, we describe the abstractions of each system using the names used by the designers in each subsystem. We then summarize the important abstractions shared across several systems in a final subsection at the end.

2.2 Component Composition Systems

The first section of our survey presents systems generally used in smaller scale ubicomp environments that provide a "lower level" abstraction for component registration, discovery, communications and composition. Unlike other categories of systems in this survey, several commercially used systems belong here: Jini, Universal Plug and Play (UPnP), and Zero Configuration Networking (ZeroConf) are discussed in Section 2.2.1. These systems can be used directly for ubicomp application development since they were designed for a changing distributed execution environment. Another commercial standard presented in Section 2.2.1 is the

Open Systems Gateway Initiative (OSGi), which provides a centralized, dynamically extensible execution environment. The Appliance Data Services (ADS) system (2.2.2), SpeakEasy (2.2.3), the Equip Component Toolkit (2.2.4), and PCOM research systems (2.2.5) all compose software components, often corresponding to devices in an environment. ADS and PCOM compose services automatically while SpeakEasy and ECT allow end users to compose device components for data transformation, display and storage. The Reflective Middleware for Mobile Computing (2.2.6) system provides a single API for several distributed component systems. The systems surveyed in this section are summarized in Table 2.1.

Table 2.1: Summary of Surveyed Component Composition Systems.

System	Purpose	Scale	Abstractions	Refs
Jini	Interoperability	Small-medium: lookup services	Object registration, discovery, services, events, transactions	[106]
UPnP	Interoperability, simple device networking	Small-medium: broadcast network	Device registration, discovery, services, state, events	[74]
ZeroConfig	Interoperability, simple networking	Small-medium: broadcast network	Service registration, discovery	[13, 29]
OSGi	Dynamically extensible service host platform	Medium: room-house	Bundle (object collections) registration, discovery, services, state, events	[80]
Appliance Data Services	Task based composition of services for data transformation/-transfer	Small: collection of devices	Tasks as sequences of service executions on data hosted in infrastructure	[60]
SpeakEasy/ Obje	End user composition of devices/components for data transfer	Small: collection of devices	Objects with generic interfaces for data transfer, grouping, metadata and UI	[76]
Equip Component Toolkit	Easy end user composition of components for control and data processing	Small: collection of devices	Components contained in a DataSpace linked using compatible properties/state	[47, 61]
PCOM	Automatic and dynamic composition of services	Small-medium: broadcast network	Services composed dynamically using dependency contracts	[17, 18]
ReMMoC	Interoperability with various service infrastructures	Small-medium:: broadcast network	Abstract service discovery, reflection and interaction interface	[45]

2.2.1 Commercial Systems and Standards

Commercial systems described in this section have been developed to create applications that can dynamically adapt to a changing network environment: a key requirement for ubicomp. Most expose the ability to communicate with individual software components that may come and go to get and change state, subscribe to events, and call services.

Jini. Jini was designed to provide a way for distributed objects to find each other and work together on a network. Its use of mobile code and support for dynamic service registration and discovery make it an obvious starting point for ubicomp systems. A fixed *Lookup Service* is deployed on a network for lease-based component registration and lookup. A client will typically search for a Lookup Service using multicast, then find the service it needs using a template-based search. The proxy code needed to communicate with the service is downloaded from the Lookup Service to communicate with it directly. Clients can register for asynchronous event notifications by registering their own remote service interface.

Universal Plug and Play (UPnP). The purpose of UPnP is to make it easy to set up and configure networked devices such as printers, Internet gateways and consumer electronics without the need for specialized servers or other infrastructure. There are three building blocks of a UPnP system: a *Device* is a container for services and other nested devices, *Services* are units of control consisting of state variables, implementations of actions, and an event server implementation, to publish events to subscribers when its state changes. Finally, applications as *Control Points* discover and control other devices by invoking services, subscribing to events, and getting device and service descriptions. A protocol is used for devices to announce themselves and for control points to find resources, no discovery server is needed; every control point maintains information on the devices

available to minimize traffic.

Zero Configuration Networking (Zeroconf). Zeroconf, also known as Bonjour and Rendezvous [13] is a collection of standard and proposals used by Apple for dynamic discovery of devices and services on IP networks. These specifications include dynamic link local addressing, multicast domain name system (DNS) for use in small networks such as the home, where there is no conventional unicast DNS server, DNS based service discovery, DNS related notifications, and protocols for printing and file sharing. Unlike Jini and UPnP, Bonjour does not include specific mechanisms and protocols for service interaction, applications use any protocol on top of TCP or UDP.

OSGi. The Open Services Gateway Initiative Alliance defines a Java based platform that defines a dynamic life cycle model for modules called *bundles*, which can be remotely installed and started without rebooting the system. It also includes a service registry to detect the addition and removal of services for applications to act accordingly. Many layers are built on this core including services for logging, administration, security, and integration with systems such as UPnP. Although it was originally designed for service gateways, it is now used in a wide range of applications such as integrated development environments, enterprise application servers, cars and mobile phones.

2.2.2 Appliance Data Services

The designers of ADS [60] recognized that the use of distributed services and devices such as those supported by Jini or UPnP often revolves around moving data from one to another. Unfortunately, in some cases, this task can be unexpectedly difficult. ADS addresses this problem by allowing developers to define tasks such as “move photos from camera to online photo album”. ADS applications are a

composition of services that operate on data or content supplied by devices. The framework consists of three components: *Data Receive*, *Application Control* and *Service Execution*. Data is received from a device, and put into the infrastructure, a shared data store, by the Data Receive components. Application Control components determine the task to perform by looking at a user id and requested command to find a matching template listing the services required. The Execution Component executes services in turn to operate on the data supplied in the receive stage or output by a previous service as defined by control template. A key abstraction introduced by ADS is the notion of a service composition they call an *application* and the support for *data* as an abstraction in the system passed between services.

2.2.3 SpeakEasy/Obje

Speakeasy researchers decided to take a different approach from ADS to moving data between devices. Unlike the ADS system, SpeakEasy allows *end users* to opportunistically assemble devices for data exchange, even when the devices do not support the same data protocols [76]. Unlike ADS which relied on compatible service interfaces for composition, the system uses mobile Java code to translate potentially incompatible data streams between components and to supply interfaces for end user control. Every component, typically corresponding to a physical device or software service on a local network, exposes one or more of a small set of generic interfaces. These interfaces are used to (1) transfer data, (2) group related components together (3) reveal and use metadata about the component, and (4) allow end user control with a user interface. SpeakEasy component interfaces allowed end users to easily compose and control devices for data transfer and processing.

2.2.4 Equip Component Toolkit

While components in Speakeasy communicate with each other directly using generic interfaces, the Equip Component Toolkit exploits the use of a shared, distributed *Dataspace* to create component compositions. The Dataspace contains references to software components such as JavaBeans, the current value of their properties, and links between compatible properties. Applications are created by "wiring up" compatible properties, so that when the value of one property in a component changes, its value is relayed to the property in another component. End users and developers can instantiate and configure components in the Dataspace using a graphical editor such as the supplied GraphEditor, or the Jigsaw puzzle editor [61]. A key advantage of the system is the ability for components to interoperate by connecting properties; no common service API or mobile code is required.

2.2.5 PCOM

In mobile applications, the services available in an environment may come and go over time. To use a commercial system like Jini or UPnP, applications must adapt themselves to the changing resources they need at run time. Another approach is for a system like PCOM to relieve applications from this responsibility. PCOM was designed to adapt automatically to changing protocols, and the services available. It assembles service hierarchies dynamically based on application-programmer defined service dependency contracts. Application programmers provide service requirements (events and service interfaces) in advance. PCOM then creates a hierarchy of services dynamically to fulfill the contracts using the current and changing service execution environment. Application programmers can use built-in PCOM adaptation mechanisms or provide their own strategies to reselect or discontinue required components at run time as the environment changes.

2.2.6 Reflective Middleware for Mobile Computing (ReMMoC)

Similar to PCOM, ReMMoC addresses the problem of clients finding and making use of appropriate services in ubiquitous computing environments. However, unlike PCOM, ReMMoC makes use of existing service infrastructures directly. In such environments, different service discovery protocols may exist such as Jini, UPnP or ZeroConf. In addition, different interaction protocols such as Simple Object Access Protocol (SOAP), or Common Request Broker (CORBA) Internet Inter-Orb Protocol (IIOP) may be used. The ReMMoC mobile device middleware uses a pluggable component architecture to address both service discovery and interaction heterogeneity. A generic service lookup interface hides the details of different service discovery protocols. A generic binding abstraction, based on Web Service Definition Language (WSDL) allows for the abstract definition of any service independent of the underlying service provider.

2.2.7 Discussion

Systems in this category revolve around the use of distributed components exposed using a service abstraction. The ReMMoC system provided a generic service API for systems such as UPnP and Jini. Several systems in this category also provide access to component state or properties, and asynchronous events or service callbacks. In several of these systems, applications are considered to be compositions of components composed or connected to perform some task. For example, ECT developers can compose components using a Dataspace that encapsulates all components and their property links in a deployment. PCOM composes services automatically and dynamically as the execution environment of an application changes, without centralized infrastructure. The ADS system introduced a data abstraction to pass between services in predefined compositions. SpeakEasy users compose

devices on the fly to transfer and process data. To summarize, these systems typically support the composition of components that expose *service*, *state*, *events*, and *data* or *state* abstractions.

2.3 Context Frameworks

This category lists those systems designed to make use of *context*. Dey defined context as "any information that can be used to characterize the situation of entities (i.e. whether a person, place or object) that are considered relevant to the user and an application..." [35]. In the simplest sense, context is some state associated with a physical or electronic entity. Many of the systems here such as ParcTab (Section 2.3.1), the Context Toolkit (2.3.2), and the Java Context Aware Framework (2.3.5), include abstractions to aggregate such information about people, places or things. Several systems such as Sentient Objects (2.3.4) can infer higher level context information such as user activity and the local weather conditions from raw sensor data, external information sources combined with lower level context information such as location and time. For the purpose of this survey, we consider *entities* and *context* abstractions exposed by these systems to be "medium" level ubicomp abstractions. The systems listed in Table 2.2 are typically designed for larger installations than Component Composition Systems since they often aggregate information and the capabilities of individual components.

2.3.1 ParcTab

ParcTab was a pioneering ubicomp system that leveraged the Remote Procedure Call (RPC) model for distributed middleware to allow applications hosted on workstations to interact with users with mobile devices. In later work, Schilit et al. [91] highlighted the need for dynamic customization as a key concept common to ubi-

Table 2.2: Summary of Surveyed Context Frameworks.

System	Purpose	Scale	Abstractions	Key Refs
ParcTab	Application customization using state about users and locations	Medium: multiple rooms	User agents and sensors supply state to Environment Servers corresponding to entities: users and locations	[90, 91]
Context Toolkit	Easy context aware application development	Medium: multiple rooms	Discoverer for component lookup/registration, widgets that support context query, events, interpreters and entity aggregators	[36]
one.world	App. changing execution environment, sharing data between devices	Small: broadcast network	Environments, contain data and components (typically associated with entities), asynchronous events for all communications	[48]
Sentient Objects	Context-aware application development in ad hoc network environments	Small-Medium: ad hoc networks	Distributed sentient objects (typically associated with entities). Event based communication, framework for context reasoning, interpretation, & aggregation	[19]
Java Context Aware Framework	Standardize Java context aware application APIs	Medium: distributed servers	Entities associated with context items that may also be other entities.	[15]

comp applications. They describe a system made up of a collection of *environment servers* each corresponding to users, places, workgroups and other entities. Each environment server maintained a set of names and values corresponding to some relevant information about the physical or computational run time environment. A *user agent* updated environment servers on behalf of the user it served. Applications typically monitored for changes in the environment by subscribing to variable changes on environment servers. An application could subscribe to the environment server for a group of users, and to the servers corresponding to the locations of these users for example. The servers used by a given application could change over time as a users location changes. An important abstraction from this early work is the environment server corresponding to a place, user, group or other entity. These servers aggregated relevant state about an entity called *context* in

follow on systems.

2.3.2 The Context Toolkit

The Context Toolkit aimed to provide a set of abstractions for the rapid development of context aware applications. A typical deployment contained a number of self describing distributed components on a LAN that supply context (e.g. sensor data) and/or services to applications. These components register with a centralized *Discoverer* which maintains information for lookup by applications or other components. The component types included *Context Widgets* which can be queried directly or subscribed to using asynchronous events, *Context Interpreters* used to translate one form of context to another, for example from an RFID tag reading to a user name. *Entity Aggregator* components were used to aggregate context and services around an entity: a person, place or object. *Services*, typically implemented by Context Widgets were exposed to interact with software services and actuators. Recognizing that applications still had to do a lot of work to find and interact with the various component types provided by the toolkit, the system was extended to provide a higher level, *situation* abstraction: a collection of relevant context queries (called the situation) in a single interface that deals with multiple components. The Context Toolkit situation abstraction forshadows the ability to model whole environments demonstrated by Smart Space Systems in Section 2.4.

2.3.3 One.world

One.world was designed for ad hoc composition and data sharing between applications and devices in a changing execution environment. To address this need, the designers argue that data and functional abstractions, unlike objects in object oriented systems that combine these abstractions, should be separated to facilitate

data sharing, searching and filtering. To establish this separation, they created an *environment* abstraction as a way of structuring and composing applications. Environments serve as storage for shared data using tuples, and containers for application components and other environments in a hierarchical fashion. Components in an environment communicate with each other using asynchronous events. Generally, environments correspond to entities such as people, places or objects and can migrate from one device to another as a user moves for example. Like ParcTab environment servers, and Context Toolkit Entity Aggregators, one.worlds environment abstraction acts as both a container and aggregator for related entity state and functionality.

2.3.4 Sentient Objects

The Sentient Object Model was developed for context-aware application development in ad hoc network environments. Like one.world, the designers anticipated a degree of mobility, and resilience to changes in connectivity between components. This model defines abstractions for sensors and actuators and a framework for creating *Sentient Objects*. Sentient objects retrieve information about their environment from each other using event-based communications [19], or directly from sensors. Sentient Objects work independently, and proactively try to achieve goals and anticipate problems. A framework for creating Sentient Objects makes it easier for developers obtain, aggregate and interpret context information received by the object. This framework includes probabilistic reasoning capabilities to interpret raw sensor data, and to derive higher level context from lower level sub contexts in a hierarchical fashion. An entity such as a person, place or thing typically corresponds to one or a group of sentient objects in a deployment.

2.3.5 Java Context Aware Framework

The Java Context Aware Framework (JCAF) is a more recent system used for context aware application development. Inspired by previous work like the Context Toolkit, a deployment consists of context services that receive, manage, store and distribute context information for one or more entities. The programming model for a *context service* consists of *entities* associated with *context items* using a context relation. Context items may also be other entities so that useful entity relationships can be established. Context clients typically access entities and their context using a context service, by registering interest in events associated with specific context to receive notifications, or querying for a context value at any time. Clients can also be suppliers (called *context monitors*) or consumers (*context actuators*) of context information from context services which aggregate the context for one or more entities. JCAF's programming model refines the high level abstractions found in other frameworks consisting of event based asynchronous communications, entities, context, and entity relationships as a specialization of context.

2.3.6 Discussion

The systems in this category support the development of *context aware* applications: applications that use relevant information about the user and their situation. These systems often contain and/or aggregate access to relevant information about people, places and things around an *entity* abstraction. ParcTab designers call these entity aggregators *Environment Servers*; they are *Entity Aggregators* in the Context Toolkit, *Environments* in one.world, *Sentient Objects*. *Context services* contain entities in the JCAF programming model. Several systems allow applications outside of the framework to query for context on demand, and subscribe to changes in context values. Sentient Objects and one.world host applications within their entity

containers (i.e. sentient objects or environments). In some cases context values can be other entities, hinting at an important specialization of context we call *entity relationships* found to be valuable in Wide Area systems described in Section 2.5. To summarize, the systems in this category build on Component Compositions to introduce entity aggregations, and context abstractions to application developers.

2.4 Smart Space Systems

In this section we consider the systems listed in Table 2.3 that typically provide a higher level of abstraction than Context Frameworks. Generally, the scale of deployment in this category are comparable to those in Context Frameworks: single rooms, or buildings. The core of both the iROS (Section 2.4.1) system and Gaia (2.4.4) is a centralized event broker to move messages between distributed components. Other core components in Gaia and iROS were created for data storage and transformation, shared environment state and services. InConcert/EasyLiving (2.4.3) and Sentient Computing (2.4.2) efforts focused on providing a detailed environment model to applications to customize interaction based on the user's absolute and relative location to other objects and people. We also consider ontology-based systems in this category in Section 2.4.5. The use of ontologies and a knowledge base in these systems allows the semantics of objects such as entities, relationships and context values to be standardized for interoperability. Information in the knowledge base can be processed using standard semantic web reasoning systems to infer new context values and trigger application services.

2.4.1 iROS

iROS aimed to make it easier to create applications for a specific class of smart space: meeting rooms. In these scenarios it is important to support multi-device

Table 2.3: Summary of Smart Space Systems.

System	Purpose	Scale	Abstractions	Key Refs
iROS	Meeting room application development	Medium: room	Centralized Event Heap, shared state, ICrafter services, Data Heap for storage and transformation	[42, 63, 83]
Sentient Computing	Easy location-aware programming	Medium: floor	Detailed environment model containing entities, absolute and relative location facts, and location events	[11, 54]
InConcert/ EasyLiving	Dynamic cross device (screens) user interfaces based on location	Medium: house	Detailed environment model containing fixed and mobile entities and their geometric relationships.	[25]
Gaia	General purpose smart space operating system	Medium: room-floor	Centralized event broker, data store, service infrastructure, space/presence repository, and context inference service	[87]
Ontology-based: SoCAM, CML, Gaia, CoBrA	Easy context aware smart space application development and implementation independent model	Medium: room-floor	Entities and context in an executable knowledge base, application execution chosen or triggered using rules.	SoCAM [51] CoBrA [28] Gaia [85] CML [55, 56]

interactions where users can move between a portable device such as a PDA or Tablet PC to one or more large wall-mounted displays. Recognizing the value of the *event* abstraction in interactive desktop applications, the main subsystem of iROS is the centralized *Event Heap*. Using the Event Heap, any device can produce events, and any number of event consumers can listen, enabling group communications and multi-device interaction. The *State Manager* subsystem makes use of the Event Heap to maintain shared state of devices, and software components in the room. Shared state includes published service descriptions used by the *ICrafter* subsystem [82]. ICrafter provides a service discovery and interaction interface similar to the Context Toolkit Discoverer and Service components. The *Data Heap* was used for storing content and documents, and meta-data associated with this content. Notably, this system also provided data-format transformation services for applications to make it easier to view and manipulate content on a wide

range of devices. To summarize, all of the devices in a room use centralized servers for event communications, shared state and data storage.

2.4.2 Sentient Computing

Unlike iROS, the Sentient Computing platform was designed to provide an explicit model of the physical environment for applications. The system provides a very fine grained location system [107], and a detailed data model of space for event based applications. To build and maintain this model, components called *resource monitors* and *spatial monitors* are used. The detailed environment model describes the entities (people, places, and things) and possible ways of interacting with them. Applications are provided with an API for location-aware programming providing both absolute and relative location facts, such as "the user is at (*x*,*y*) facing direction (*angle*)", or "the person (*Bob*) is standing in front of workstation (*Xyz*)". Sentient Computing highlights the value of an explicit, accurate and dynamic environment model to ease application development. Another important distinction from Component Composition Systems and some Context Frameworks is that applications need not access components that implement or maintain the model since the system itself effectively hides these concerns.

2.4.3 InConcert/Easy Living

EasyLiving designers focused on supporting user interaction across multiple devices including mobile devices and large screen displays in the home. Like Sentient Computing, the InConcert middleware provides a explicit environment model to enable applications to dynamically assemble a user interface across multiple devices. Knowing a user's location and orientation allows the system to choose which display to use for information, and which speakers to use for music or voice

responses. Within this environment model, objects in the physical world are associated with each other using geometric entity relationships called *measurements*. As in Sentient Computing, and Gaia discussed next, software tracks mobile objects to maintain the model so that these relationships are kept current. Developers make use of the geometry model and service descriptions to adapt their user interface to the current situation. Maintaining the measurements using the geometry model is a key enabler for EasyLiving applications that must take into consideration the spatial relationships between devices and end users in the environment.

2.4.4 Gaia

Gaia aimed to apply and extend approaches proven in systems like iROS and Sentient Computing to domains such as the home, the office, and the car. Like iROS' Event Heap, a centralized event broker called the *Event Manager* provides a publish-subscribe mechanism for services, applications and components. The *Space Repository* and *Presence Service* subsystems store and track tracks the physical location of entities, such as people and devices and software components for applications. A general purpose *Context Service* tracks other context such as sound, temperature and weather. This subsystem supports the use of first order logic to infer higher level context from sensor data. A *Context File System* allows users to associate content with different contexts such as time, place, and user presence. Finally an *Application Framework* made it easier for application developers to make use of the various Gaia subsystems and distributed components. Since interacting with multiple devices can be challenging, the notion of an "Application Session" was created for end users to contain the applications and data associated with that user when they enter the space.

2.4.5 Ontology Based Systems

Several systems in this survey demonstrate that that application code can be reused with the consistent use of an interface to a given smart space. However, to address application portability and interoperability between smart spaces, it is not only important to share the same interface, but also the same semantics for context types, and service interfaces. In ontology-based context-aware systems such as SoCAM [51] and the Context Broker Architecture (CoBrA) [28], context is modeled with a model called an ontology, a formal description of concepts in a particular domain. Gaia was also extended to use ontologies for context reasoning [85] about entities and components in the system. The Context Modelling Language (CML) [55, 56] is a graphical notation developed to assist developers to design and explore the context requirements of applications independent of the infrastructure used. While the CML does not use sematic web notations or technologies such as the Web Ontology Language (OWL) [102], it does relate attributes to physical and conceptual entities such as users, and devices. CML also permits designers to specify context quality and dependencies between context information.

In ontology-based systems, logical expressions using facts in the knowledge base can define situation abstractions similar that in the Context Toolkit, inferring higher level context from lower level facts in the environment model. The use of an ontology provides a way to share common understanding of concepts in an environment and facilitates the use of an executable model in a knowledge base. An executable model with a general purpose reasoning engine can also be used to infer higher level context and new entity relationships from facts in the model, or to alter application behavior and execute specific services when certain situations exists.

2.4.6 Discussion

Smart Space Systems aim to provide even more comprehensive support for application development in specific places such as meeting rooms, and the home. Typically these systems coordinate multiple mobile devices, users, and large fixed displays: inter-device interaction is a critical requirement. iROS and Gaia centralize communications between components to broadcast and intercept the value of *events* as a key abstraction for smart spaces. Several systems such as EasyLiving and Sentient Computing introduce the notion of a comprehensive and explicit *environment model*. Ontology based systems maintain information about contained entities, their relationships to one another, and current context values in a central knowledge base. All of the systems provide mechanisms to call *services* independent of their underlying implementation and location in the smart space. iROS and Gaia both provide comprehensive *data and content storage* and transformation services for applications. To summarize, smart space systems highlight the value of cross-device interaction events, explicit environment models and centralized data and data transformation services for application developers in their programming models.

2.5 Wide Area Systems

The systems listed in Table 2.4 coordinate larger scale environments containing many users and applications potentially across smart spaces. The cross-physical, -network and -administrative domain requirements for such systems means that many of the communications protocols used in other categories are not appropriate; the systems here use protocols such as HTTP and Web Services rather than local network broadcast for example. Security and privacy are also a consideration in this

category. Since the number of computing resources in large scale environments can be high, and communications can be expensive in terms of latency and bandwidth, there is often a need to provide coarser grained abstractions than software components, individual devices and services in this class of systems. Typically these systems will aggregate information and services associated with entities: people, places and things, or environments as a whole as in Context Frameworks or Smart Space Systems. The first system we describe in Section 2.5.1, Cooltown, leveraged the well understood distributed document model of the web to integrate the physical world with the online world. The Context Fabric (2.5.2) aimed to provide a privacy sensitive context infrastructure by linking distributed Infospaces containing context about entities. Active Campus (2.5.5) provided integrated large scale ubicomp environment containing many services to potentially thousands of users on a university campus. The Nexus (2.5.3) system designers federate environments, while Aura (2.5.4) aimed to support migration of high level user tasks between smart spaces by marshalling the services there. Web Service for Ambient Intelligence (WSAMI) in Section 2.5.6 is unique in that it composes wide area services taking a similar approach to Component Composition Systems.

2.5.1 Cooltown

The World Wide Web introduced a model for distributed computing where information is organized into documents identified by uniform resource locators (URL) linked to other documents elsewhere in the world. Cooltown leverages this simple and effective model for ubicomp by providing a software layer to integrate the physical environment with the web [68]. People, places and things in the world each have their own web presence, software running on a server that provides a web user interface to an entity. The web presence of a user is related to other

Table 2.4: Summary of Wide Area Systems.

System	Purpose	Scale	Abstractions	Key Refs
CoolTown	Leverage WWW to integrate physical world with online world	Large	Web presence servers corresponding to entities (people, places, objects) linked to each other	[68]
Context Fabric	Context infrastructure for privacy sensitive applications	Large	Network of Infospaces typically corresponding to entities containing instrinsic and extrinsic context (entity relationships)	[58, 59]
Nexus	Infrastructure for spatial-aware applications	Large	Network of Augmented Areas containing entities (objects of interest) in an Augmented World. Location queries and events.	[57, 77]
Aura	User/task migration between smart spaces, and changing devices	Multiple-smart spaces	Task abstraction to marshal end-user services (Suppliers) in the environment based on context information in an Environment Manager	[94]
Active Campus	Address tradeoffs between extensibility and integration in large scale ubicomp	Large: campus	Two layer environment model associating entities with services and context. Entity Modeling layer deals with static relationships, Situation Modeling layer with dynamic context and relationships.	[49, 50]
WSAMI	Situation aware web services composition	Large: independent of location	Web services composed dynamically using dependency and QoS requirements.	[3, 62]

entities such as places, or objects nearby using dynamic web links corresponding to directed entity-entity relationships. These relationships may include *contains*, *next-to*, or *carried-by* for example; they may be reciprocal or one-way relationships to protect user privacy. Users typically interact with one web presence at a time, starting with their personal (user) web presence; sensors are used to discover new entities and dynamically create entity links. For example, when a person enters a room, the users web presence is linked to the rooms web presence when an infrared beacon is detected by a mobile device carried by the user. Users (and applications) may follow the links to take advantage of functionality (applications) hosted by the rooms web presence. A key abstraction for Cooltown is the notion of *entity relationships* corresponding to web links used by users and applications

hosted on web servers to adapt to a changing environment.

2.5.2 Context Fabric

The Context Fabric was proposed as an infrastructure for context storage and management. When collected and distributed in a shared infrastructure, privacy of context information such as location and activity is a vital concern for end users. To address this, context can be manipulated (e.g. aggregated or anonymized) as it enters or leaves the system to manage the privacy requirements. The Context Fabric is a network of servers containing *InfoSpaces* corresponding to entities such as people, places and things. An InfoSpace contains context about the entity it handles, both *intrinsic* context, information about the entity itself, and *extrinsic* context, relationships between entities. Applications can also use the InfoSpace to store service descriptions. A client library simplifies querying by supporting on demand, periodic and subscription based queries on InfoSpace data. The Context Fabric is similar to several systems in the Context Frameworks category and Cootown in that it aggregates context data in Infospaces corresponding to entities. Infospaces aggregate context about entities, and relate them to one another, similar to Cooltown web links.

2.5.3 Nexus

Nexus is a generic wide area infrastructure for location and spatial-aware applications [57]. The designers aimed to provide a model of regions of the physical world called *Augmented Areas* similar in concept to the explicit environments in Smart Space Systems such as EasyLiving or Sentient Computing. Both physical and virtual *objects of interest* exist in Augmented Areas accessible through the platform. An object of interest may also be a proxy for an end user. Furthermore, Augmented

Areas can be federated in a global *Augmented World* related to one another by containment and relative distance relationships. Since all Augmented Areas use the same Nexus interface, applications can easily move between Augmented Areas using a handoff mechanism. An important attribute for objects in an Area is location determined using an Active Badge or GPS for example. Like Sentient Computing, the system supports location events triggered by changes in location such as entering areas or proximity to other users, but aims to unify such smart space capabilities for larger scale deployments. Nexus demonstrates that federating smart spaces is another approach to addressing scalability and smart space integration.

2.5.4 Aura

In the Aura system, user mobility between ubiquitous computing environments is supported with an abstraction called a user's personal *aura* that encapsulates the users current task. Similar to the notion of an Application Session in Gaia, the aura or *task* is defined as the information and services (applications) required by a user at a given time. Once the users task is transferred to an instance of the system, the hub of the system, called the *Task Manager* marshals resources in the current environment to support that task. Services are hosted by components called *Suppliers* which register with the Task Manager. A service hosted by a Supplier may be a display device, a text editor, storage server or drawing application for example. Like Gaia's Space Repository, Aura's *Environment Manager* manages information related to the physical environment. Aura uses a *Context Observer* to watch the environment for end user activity, to report this information to the Environment Manager. When the user moves from one environment to another, that user's Aura can be migrated to the Task Manager at the new location to continue their current task. Each Aura system provides an environment model for task execution and

demonstrated that the use of a common platform in multiple smart spaces makes it possible to for users seamlessly move between environments, migrating data and applications on their behalf.

2.5.5 ActiveCampus

Like Cooltown, Active Campus services are presented as web pages, however, the goals of this system are quite different. This project aimed to address the trade-offs between easily providing new services, while maintaining the integration of these services in an large (campus) scale environment [49]. The centralized ActiveCampus architecture consists of several layers. In the top layer, mobile devices communicate with an *Environment Proxy* which marshals data between the services on the device and the ActiveCampus system. The *Situation Modeling* layer in the server synthesizes the situation of entities from multiple information sources (mobile devices and other sensors). Finally, the *Entity Modeling* layer of the server represents entities in several forms for access by other services and presentation on a browser, and stores static relationships among these entities. In this system we again see an explicit environment model containing entities, entity relationships and entity-related context. Unlike Context Frameworks and Smart Space Systems, however, ActiveCampus was designed to scale to larger environments and more users by making use of a layered architecture.

2.5.6 Web Services for Ambient Intelligence (WSAMI)

WSAMI [62], part of the Ozone project [3], is a middleware that leverages Web Services standards to deploy and compose web services dynamically on wireless networks and mobile devices. The use of Web Services enables availability in most environments, and potentially across network domains. WSAMI uses a *nam-*

ing&discovery service that supports naming, service discovery and lookup in both local and wide area networks. Like PCOM and other Component Composition Systems described in Section 2.2, this work focuses on dynamic distributed service composition. Unlike these systems, however the designers targeted wide area web service composition. Novel aspects of this work included the customization of the network links for performance and security. Services are specified using standard WSDL. These specifications are then referred to in WSAMI specifications which include the required services that an application needs.

2.5.7 Discussion

We find that systems in the wide area category tend to borrow abstractions from the other three. For example, WSAMI exposes and composes services directly like others in the Component Composition category. The Context Fabric exposes entities and context, like other Context Frameworks. The Nexus system federates whole environments called *Augmented Areas* similar in concept to the environment models exposed by Smart Space Systems.

For greater scalability large scale systems will often distribute storage processing among multiple servers. Several systems here distribute work among servers that proxy physical or virtual entities in the real world. These servers will often expose relationships between servers that correspond to their proxied entity's real world relationships. The Context Fabric, for example, aggregates information about entities in Infospaces, relating them to each other using *extrinsic* context. Similarly, Cooltown aggregates services and relevant information about entities using separate web servers, relating them to each other using hypertext links.

Unlike other systems, the Active Campus system addresses scalability by separating the concerns of managing an integrated environment model into separate

layers on a single server. Aura is also unique in that it supports user migration between environments by migrating information about a users current task to be transferred from one smart space to the other.

2.6 Common Abstractions Derived from the Survey

Based on this survey, two things become evident: one is that we have a wealth of experience to draw from when designing new ubicomp systems. Secondly, we see that certain high level concepts are shared by several systems, in some cases, under different names. Systems in all categories, for example, supply *services* to applications; functionality exposed using an interface registered with the system. Systems in all categories support application callbacks or *events* for notification when something about the state of a device, entity or the environment changes. Context Frameworks tend to aggregate components and information around an *entity* abstraction to avoid the need to communicate with multiple components to find *context*, relevant information about that entity [36]. Several smart space systems illustrate the value of a centralized and explicit *environment model* containing detailed information about entities, context and relationships. Several component compositions expose a *data* abstraction and certain smart space systems support data or *content* storage. We note that Wide Area Systems surveyed here often use real world *relationships* between entities: people, places, things, and whole environments to link servers representing these entities. This allows applications to easily find and use relevant context and services, avoids unecessary communications to individual components, and lends itself to greater scalability by distributing storage and processing of context information. Table 2.5 indicates when one of these abstraction appears in a given system.

Table 2.5: Summary of the scale, level of abstraction and abstractions used. Here we indicate whether an abstraction is supported (✓), not supported (✗) or partially/implicitly supported (P).

System	Abstraction Level	Scale	Env. Model	Entities	Context	Entity Rel.	Services	Events	Data / Content
Jini	L	S	P[a]	✗	P[b]	✗	✓	✓	✗
UPnP	L	S	P[a]	P[c]	P[b]	✗	✓	✓	✗
ZeroConfig	L	S	P[a]	✗	✗	✗	✗	✗	✗
OSGi	L	S-M	P[a]	✗	P[b]	✗	✓	✓	✗
ADS	L	S	P[a]	✗	✗	✗	✓	✗	✓
SpeakEasy /Obje	L	S	P[a]	P[c]	P[b]	✗	✗	✗	✓
ECT	L	S	P[d]	✗	✗	✗	✗	✓	✗
PCOM	L	S	✗	✗	✗	✗	✓	✓	✗
ReMMoC	L	S	✗	✗	✗	✗	✓	✗	✗
ParcTab	M	M	✗	✓	✓	✗	✗	✓	✗
Context Toolkit	M	M	P[a]	✓	✓	P	✓	✓	P[e]
one.world	M	M	✗[f]	✓	✓	P[g]	✗	✓	✓
Sentient Objects	M	M	P[h]	✓	✓	P[i]	✓	✓	✗
JCAF	M	M	✗	✓	✓	✓	✓	✓	✗
iROS	H	M	P[j]	P[k]	✓	✗	✓	✓	✓
Sentient Computing	H	M	✓	✓	✓	✓	✓	✓	✗
EasyLiving	H	M	✓	✓	✓	✓	P	✓	✗
Gaia	H	M	✓	✓	✓	✗	✓	✓	✓
Ontology Systems	H	M	✓	✓	✓	✓	P[l]	✓	✗
CoolTown	M	L	✗	✓	P[m]	✓	P	P	P
Context Fabric	M	L	✗	✓	✓	✓	✗	✓	✗
Nexus	H	L	✓	✓	✓	✓	✓	✓	✗
Aura	H	M-L	✓	P	✓	✗	✓	P	✓
Active Campus	H	L	✓	✓	✓	✓	✓	P	P
WSAMI	L	H	✗	✗	✗	✗	✓	✓	✗

[a] a component/service or device registry
[b] component state as context
[c] Devices only
[d] Dataspace
[e] data or content treated as context
[f] although called an *environment*, typically corresponds to an individual *entity*
[g] environment/entity containment relationships
[h] the ad hoc network
[i] implicit in event subscriptions between objects
[j] Event heap, service registry
[k] embedded in event fields
[l] some ontology-based systems
[m] within web presence server

2.7 Conclusions

In this chapter we presented a survey of twenty one representative research systems, and four commercial standards used for ubicomp application development. To consider related systems together, we grouped systems into four categories: Component Composition Systems, Context Frameworks, Smart Space Systems, and Wide Area Systems. These categories were created by placing systems in a space defined by two dimensions: the typical *scale* of deployment targeted by their designers, and the *level of abstraction* they support. We defined three abstraction levels: services and components (low), entities and context (medium), and explicit environment models (high). By considering groups of systems in this space it becomes evident that there is some correspondence between the scale, and the level of abstraction that system exposes. Small scale systems tend to expose simpler service/component abstractions while larger scale systems provide additional abstraction layers, alleviating the need for applications to find individual components and model physical environments themselves. We noted also that some systems demonstrate that these dimensions are also orthogonal. Smart Space Systems tend to provide explicit environment models (high level abstractions) to medium scale deployments (e.g. [11]), while some large scale systems provide entity and context (medium level) abstractions (e.g. [59]), or service and component composition (low level) abstractions (e.g. [62]). Finally, we derived a set of common abstractions exposed by several systems. These include an *environment model*, *entities*, *context*, *entity relationships*, *services*, *events* and *data* or *content*. We present these common abstractions in more detail in the next chapter.

Based on this survey, it is evident that there are tradeoffs between interoperability, scale of a typical deployment, and the level of interactivity or domain-specific capabilities that influences the programming abstractions chosen. With the wealth

of experimentation and deployment experience so far, we can now begin to understand these tradeoffs and address some the challenges the ubicomp community faces related to interoperability. Armed with the set of common abstractions identified here, we have a solid foundation for the design of a common programming model for ubicomp. In the next chapter we continue our analysis starting with the common abstractions we identified in this survey and describe the design of the Ubicomp Common Model.

Chapter 3

The Ubicomp Common Model

In the previous chapter we identified several categories of ubicomp systems, and summarized the core abstractions shared across categories. In this chapter we continue our analysis in Section 3.1 by highlighting several requirements for a interoperable model for ubiquitous systems. We then review the abstractions identified in Chapter 2 in Section 3.2 with examples, and present the Ubicomp Common Model design consisting of three related aspects: the Environment State, Meta State and Implementation providing examples of each in Section 3.3. In Section 3.4 we reflect on how the UCM addresses several requirements presented in Section 3.1. In Section 3.5 we discuss how the UCM could be extended to support the integration of different security domains and access control mechanisms. Section 3.6 outlines use cases for an *executable* UCM that can be queried and reasoned with. Finally we summarize the Chapter in Section 4.9.

3.1 Common Model Requirements

To drive our analysis, we envision scenarios where applications hosted on mobile phones connected to wide area networks interact with public ubicomp environments such as shopping malls and museums. We anticipate that application servers hosted outside of a ubicomp environment's network domain will need to make use of resources there. This can occur across a large university campus or between organizations to link smart meeting rooms for example. Based on our survey of

systems in Chapter 2, our own integration experience described in Chapter 5, and our deployment of other ubicomp systems [39, 40] we highlight the following requirements for the design of a common model for ubicomp:

Application Portability. An interoperable model should support a level of application portability between different environment *types* such as the home, the office and public places.

Environment Specialization. While portability is important, a common model must support specialization for different domains. A specialization may include subclasses of core entity types, service interfaces, context and event types specific to a location and its use. This will allow general purpose applications to work between locations while allowing deployments to provide location and domain-specific resources.

Introspection. To support both portability and specialization, A common model must support *introspection*, exposing not only the current environment state (entities and context information), but also its current *capabilities* such as the types of context and service interfaces available. This will allow applications to query and adapt to the the environment and the facilities that are available.

Separate Implementation. The model should separate exposed abstractions such as *entities* and *context* from implementation abstractions such as distributed *components* and *servers*. This separation of concerns will allow implementation independence, and dynamic binding of components to entities without application involvement. Available components can come and go, and change depending on the current context. By separating the implementation from exposed abstractions a supporting system can be designed to adapt to change.

Straightforward Mapping to Existing Systems. For ease of integration, a common model should lend itself to a relatively straightforward mapping to a variety of existing systems' abstractions. The model should support the integration

of different categories of ubicomp systems: Component Compositions, Context Frameworks, Smart Spaces, and Wide Area Systems as described in Chapter 2. This means there should be support for a wide variety of abstraction levels and scales of deployment. The model must find the right tradeoff between being suitably generic across a wide range of systems but semantically close enough to specific systems to take advantage of their unique capabilities.

Access Control and Security. When applications interact with environment resources across network domains (e.g. [41, 93]), or in unadministered ad hoc connections such as Bluetooth, access control and security issues are important considerations. In these scenarios, we cannot rely on a network administrator of a private deployment to ensure all of the applications in an environment are authorized and secure as the designers of closed ubicomp systems have. As a minimum, an intermediary that exposes resources outside the domain must provide access control to previously assumed private or closed deployments.

Executable Model. We also claim that that an integration model should be *executable*, that is, have the ability to be queried and reasoned with by applications. Support for flexible queries will allow applications to discover entities and associated services, and allow applications to determine whether their requirements can be satisfied. Support for reasoning will allow an integration platform to maintain the exposed model as its composition changes, simplify integration tasks, provide missing general purpose capabilities such as context inference, and establishing relationships between entities and components.

There remain many open research challenges in ubicomp such as the various dimensions of scalability, dependability, security, privacy, context management, application mobility and HCI that in some cases affect the programming model of ubicomp systems. However, until there is some consensus in whether or how these issues are exposed to application developers, it is likely too early to address

them all in the design of common programming model; however, we acknowledge that change must be anticipated in our core model design by ensuring it can be extended with new abstractions, and provide an example of this in Section 3.5. Of course, the addition of new abstractions will likely require additional interfaces to an implementation (i.e. in an integration gateway or standalone system).

A key challenge is to find the right balance between interoperability and suitability for cross domain access as outlined while maintaining as much of the functionality of a given underlying ubicomp system. This will necessitate the provision of a new layer of abstraction on top of the native one. Of course, the introduction of a new programming model *can* make application development more difficult, especially if it doesn't match the problem at hand. Just as different programming languages and supporting libraries support some application domains better than others, we expect that different environment models will need to coexist. Since we do not expect all local ubicomp applications to require cross-domain interaction, we need not replace an existing set of abstractions and associated APIs for native application development; we can provide an interoperable model as an alternative suitable for the basic needs of portable applications and cross domain access. To address these requirements we must base our model on the common abstractions of existing systems reviewed in the next section.

3.2 Existing Systems' Abstractions

To derive the UCM model we based our core abstractions on the analysis of representative systems presented in Chapter 2. We found the following core abstractions were shared across systems in all four categories, Component Composition Systems, Context Frameworks, Smart Space Systems and Wide Area Systems:

Environment Model. An environment model is an abstraction that contains the

current state of the environment for application access. This abstraction is most evident (and comprehensive) in smart space systems, containing entities, context values, and entity relationships. There are other examples of this abstraction in every category however. In component composition systems, and in some Context Frameworks, for example, this abstraction can be considered a component registry or lookup service. We can consider the Jini Lookup Service and the Context Toolkit Discovers to be simple environment models for example. The ECT *Dataspace* is an environment model that contains components and links between component properties. The Nexus infrastructure federates their environment models, while Active Campus encapsulates their environment model using a centralized integration server containing information about entities and integrated services.

Entity. An *entity* abstraction is used in several systems to represent or proxy a person, place or thing in an environment: either physical or virtual. Examples of implementations include the ParcTab *Environment Server*, the Context Toolkit *Entity Aggregator*, a Cooltown *Web Presence*, and one.world *Environment* abstraction. In many Smart Space Systems, the exposed environment model will contain entity abstractions, context, and relationships.

Context. Most Context Frameworks in this survey expose *context* as information related to an entity. Applications may query for this information, or register to be notified when context data changes. In some cases, the entity abstraction in a system such as a server or distributed component is used to aggregate context data or the components that supply context such as sensors. Again, the use of context is most evident in most of the Context Frameworks surveyed, and in some Wide Area Systems such as the Context Fabric infrastructure.

Entity Relationship. Several systems make use of a specialized form of context we call an *entity relationship* to mirror the relationships between physical objects, places and people in the real world. These include location-based relationships: a user is *contained-in* a place, objects are *near* each other. Relationships may include ownership, a user *owns* a device, or social relationships such as friendship or community group relationships. Entity relationships, are evident in Cooltown, JCAF, the Context Fabric, to some extent in one.world, and the Context Toolkit.

Service. The notion of a service, functionality exposed through an interface, is at the core of most of the Component Composition Systems listed here. It is also supported either implicitly by the underlying middleware, or explicitly in a service framework supplied by a system. For example, the iROS ICrafter subsystem supports RPC semantics for services implemented using the Event Heap. Similarly Context Toolkit Widgets can implement services with remote procedure call semantics.

Event. The iROS and Gaia systems highlighted the value of a centralized event broker at the core of a smart space system for loose coupling between applications and devices. Virtually all other systems including Context Frameworks such as one.world support events as a key communications primitive for components in ubicomp. To avoid polling, events are used to communicate important changes in device state, context values, and relationships in an environment.

Data or Content. Finally, we note that several systems, particularly in the Component Composition and Smart Space categories, support content or data as a separate abstraction. The Appliance Data Services system for example

stores data in the infrastructure to allow device services to transfer it from service to service. Both iROS and Gaia have centralized data storage and transformation services to make it easy to share data between software services and end users in a smart space. In some systems and deployments data or content is considered an entity, whereas in others, it is treated as context. For example, content stored in the Gaia Data Manager or iROS Data Heap is treated like other objects in the system, stored in the infrastructure and passed between services. The data itself may have context meta-data containing information about who created it, its title and current version. In other systems, content is treated as context. The Context Toolkit application called the Conference Assistant [37], the system treats presentation content and questions as context for example.

3.2.1 Core Abstractions to a Common Model

As a concrete example of how these core abstractions can be used to model a ubiquitous computing environment, we describe a "smart" campus as illustrated in Figure 3.1. Buidings contain classrooms containing students equipped with laptops, smart phones or other mobile devices. Each classroom includes a projector that can be used for presentations. The projector can signal when a slide changes so that users can follow the presentation on their laptops or mobile devices. Users can communicate with each other using messaging, and locate their friends in the campus.

In this environment the (static) location of buildings, classrooms should be made available to applications. We anticipate that the dynamic location of users (coordinates), their identity, online status and social relationships will also be important to certain applications.

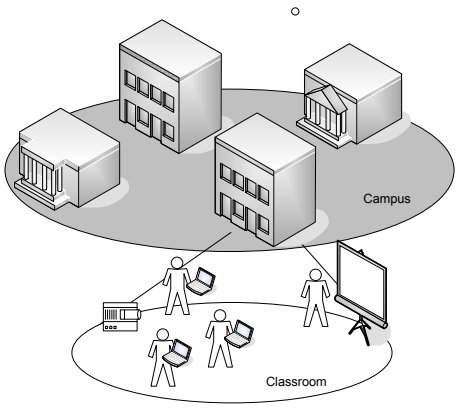


Figure 3.1: Smart campus including buildings and classrooms.

We can model this environment using the abstractions outlined as follows:

- **Environment.** The environment is a campus that hosts various entities, related to other entities, services, context and content.
- **Entities.** The campus environment hosts entities such as buildings, classroom places, users, and projectors.
- **Context.** Buildings, classrooms and users all have location context. Users have presence context (e.g. online, offline, busy).
- **Entity relationships.** Buildings and classrooms have static containment relationships. Users can be *friends* with one another, and can be *contained in* a classroom or a building.
- **Services.** In this environment, messages can be sent to users, and presentations can be made on projectors using appropriate service interfaces.
- **Events.** To keep in sync with the presentation, the projectors send events when a slide has changed.

- **Content.** A projector will have a presentation associated with it while it is being used in the classroom.

To implement the smart campus, we need a variety of software and hardware *components* to be used. An instant messaging system can provide messaging and friend management services for example. A PC can be used to drive the projector, and a location sensor in each classroom can sense the entry and exit of each user. These available computing resources will directly affect the *capabilities* exposed by the environment model. Because of this, it is important to expose these capabilities to applications so that they can adapt to their availability (or absence). By considering the three aspects of an environment: its *state* including entities, their relationships and and their current context values, its *meta-state*, the current capabilities of entities, and the *implementation* of this model, we designed the Ubicomp Common Model described next.

3.3 The Ubicomp Common Model Design

The Ubicomp Common Model is an *entity-centric* model for ubiquitous computing systems. It is entity centric in that all of a ubicomp's computing resources are related to one or more entities: e.g. people, places, things, and other physical and virtual concepts. The definition of an entity depends on the UCM specialization. To describe the UCM we used the Web Ontology Language (OWL) [102] along with a set of rules for use in a general purpose reasoning engine [1].

OWL is an ontology language built on the Resource Description Framework (RDF) [101]. Over time RDF has come to be used as a general way of modelling information and used as the basis for ontology languages such as OWL. RDF is based upon the idea of making statements about *resources* typically named by a Uniform Resource Identifier (URI). RDF statements typically provide meta-data

about those resources in the form of subject-predicate-object expressions, called *triples*. A collection of RDF triples intrinsically form a graph. In this chapter we use graphs to highlight the key concepts and relationships in the UCM as shown in Figure 3.2. Specifically we use elipses to represent OWL classes, labelled lines with arrows to represent OWL *properties*, subclasses and subproperties as shown. A rounded rectangle is used to represent properties that are also properties of a class, and rectangles for literal values (numbers and strings). In the RDF snippets in this chapter we use the XML serialization format.

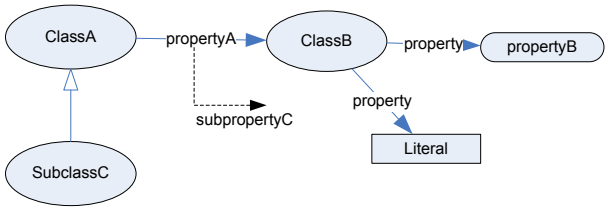


Figure 3.2: Notation used to highlight classes, properties and relationships in the UCM.

To address the requirements and scenarios highlighted in previous sections, we considered three related aspects of an environment called the *Environment State*, the *Environment Meta-State* and the *Environment Implementation* as shown in Figure 3.3. The *Environment State* aspect consists of entities, entity relationships, and the current *state* of those entities: current context values and content for example. The Environment Meta-State consists of entity *capabilities*: the types and quality of events, services, context and content an entity exposes are linked to the entities in the state aspect by the *exposes* property, (and subproperties). This aspect is necessary for application introspection. Both the Environment State and Meta-State are exposed to applications to query the current context values, entity relationships, and capabilities of an environment. Finally, the *Environment Implementation* as-

pect links entity instances by an *aggregation* property to the specific components that supply the services, context and events for a given entity. Together these aspects form the Ubicomp Common Model core.

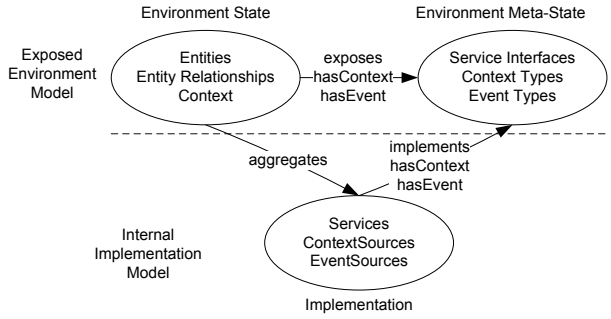


Figure 3.3: The three aspects of the Ubicomp Common Model

The three aspects of the UCM depend on one another and typically change over time. In a typical environment entities and components are added and remove and context values change. The Meta-State depends on the current Implementation, since the exposed capabilities of an entity will depend on the components aggregated by it. The current Environment State aspect depends on the Meta-State since the entities, relationships and context values of an entity will depend on its capabilities. In some cases, the components associated with a given entity will depend on the current situation; that is, the Implementation aspect will depend on the Environment State. This can occur, for example, when a mobile user changes locations, or the device they are currently using. A messaging service may change from an instant messaging implementation to SMS when the user leaves the office. The location context source may change from a GPS-based infrastructure for outdoor use to a wifi-based location sensor when indoors. We elaborate further on each aspect in the following subsections.

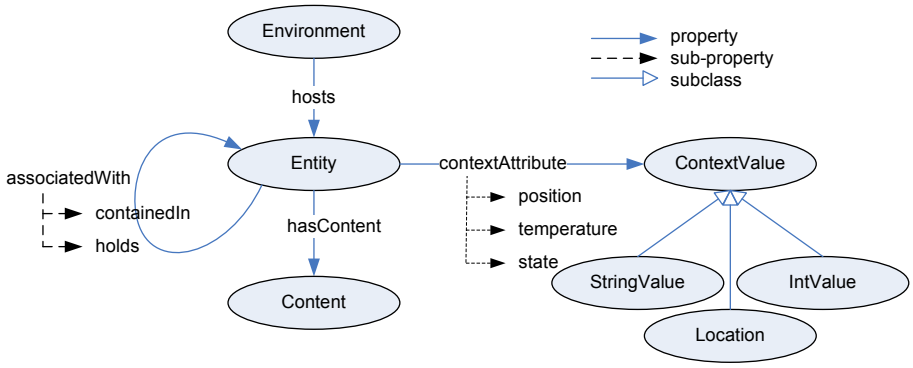


Figure 3.4: Environment State abstractions and relationships.

3.3.1 Environment State

The Environment State consists of entities modeled by the supporting system, the relationships between entities and their current context values. Context values are related to entities by context attributes. Context values need not be simple primitive types such as strings and integers, but may also be more complex data structures. These data structures could indicate a range of values, or an indication of timeliness and accuracy. The key abstractions and their relationships in the Environment State are shown in Figure 3.4.

The *Environment* object serves as the root entity of an environment and hosts other Entities and subclasses of Entities such as places, people and devices. In a supporting system context values can be retrieved by requesting the value of the associated context attribute or in an event data structure when an event is received. The *contextAttribute* property and *ContextValue* object may be specialized as shown to support different data types. Entities may also have *content* associated with them as shown by the *hasContent* relationship with a Content object.

The UCM does not define all possible context types or quality of context; rather, it is a core ontology intended for specialization by an integrator or standards

Program 3.1 Example Environment State RDF fragment.

```
<campus:CampusBuilding rdf:ID="coffeeShop">
  <location:location>
    <location:Position>
      <ucm:name>position</ucm:name>
      <ucm:javaType rdf:datatype="&xsd;string">
        ca.ubc.cs.uif.prototype.types.WorldPosition
      </ucm:javaType>
      <location:latitude rdf:datatype="&xsd;double">
        49.260537157736785</location:latitude>
      <location:longitude rdf:datatype="&xsd;double">
        -123.24801921844482</location:longitude>
      <ucm:time rdf:datatype="&xsd;long">0</ucm:time>
    </location:Position>
  </location:location>
  <ucm:containedIn rdf:resource="&campus;ubcCampus"/>
</campus:CampusBuilding>
```

group to define the context types for a specific environment or application domain. The interpretation of a given ContextValue will depend on the specialization of the UCM model for a given domain such as a campus, home, office or classroom. A fragment of Environment State RDF is shown in Program 3.1. The prefixes are a short form of the various namespaces used: *ucm* is the core UCM namespace, *location* is the namespace for a simple location ontology that extends the UCM, and *campus* is the namespace for a campus instance of the UCM. This example describes a *coffeeShop* with static location context, and a static *containedIn* relationship with the *ubcCampus* place. The *coffeeShop* has a static context *campusLocation*: a data structure containing latitude and longitude properties.

3.3.2 Environment Meta-state

The Environment Meta-State aspect is required to support introspection. It associates entities with their *capabilities*: the types and quality of events, services, context and content an entity supports as shown in Figure 3.5.

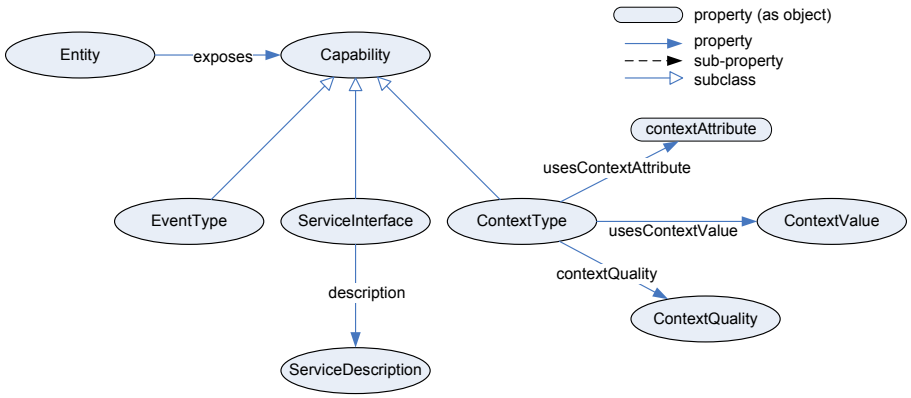


Figure 3.5: Environment Meta-State abstractions and relationships.

When an entity has context associated with it, the entity *exposes* a *ContextType* capability. A *ContextType* specifies the context attributes to use to retrieve a ContextValue (*usesContextAttribute*) and may include other properties to specify the ContextValue and quality of this context as shown. Similarly, entities may expose *ServiceInterface*s. Clients of the model can then call these services as specified in a *ServiceDescription*. *ServiceDescription*s can be specialized to support standard service descriptions such as Web Services Description Language (WSDL) [105] or others. The type of events that may be fired by an entity is specified using an *EventType* object. The RDF fragment in Program 3.2 indicates that the *CampusUser* entity *bob* exposes the *pointLocation* context type, the event type *contextChangeEvent*, and the *MessageService* interface.

3.3.3 Environment Implementation

To avoid dealing with a plethora of sensors, actuators, services, and software components, ubicomp systems typically expose a variety of distributed components, protocols or related APIs which we call *components*. The role of these compo-

Program 3.2 Example Environment Meta-State RDF fragment.

```
<campus:CampusUser rdf:ID="bob">
  <rdfs:label>Bob Smith</rdfs:label>
  <ucm:containedIn rdf:resource="#campusPlace"/>
  <ucm:exposes rdf:resource="&ucm;pointLocation"/>
  <ucm:exposes rdf:resource="&ucm;contextChangedEvent"/>
  <ucm:exposes rdf:resource="&campus;MessageService"/>
</campus:CampusUser>
```

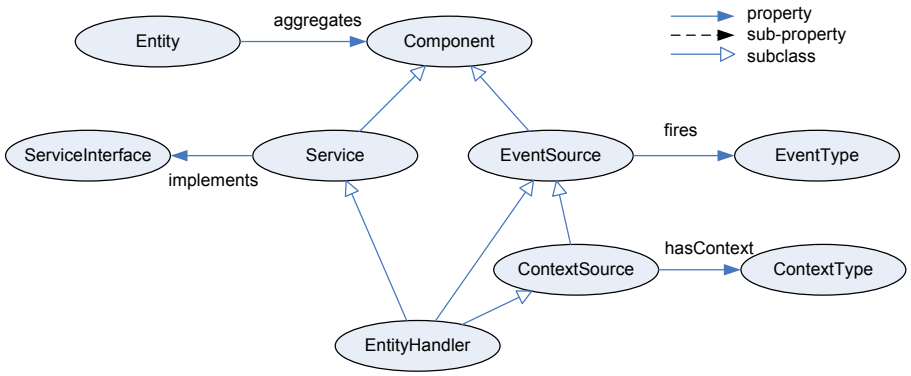


Figure 3.6: Environment Implementation abstractions and relationships.

nents are captured in the *Implementation* aspect of the UCM as shown in Figure 3.6. Here we show that components are aggregated by an entity instance and form a class hierarchy. Service components *implement* a *ServiceInterface*, *EventSource* components *fire EventTypes*, and *ContextSource*s have a *ContextType*. These component abstractions can be used to map the common model to corresponding APIs in an existing system to invoke services, retrieve context or fire events. When an entity aggregates a component, the following rule ensures that that entity exposes the types and interfaces that component implements in the Environment Meta-State.

```
[aggregateComponent:  (?entity ucm:exposes ?capability) <-
    (?entity ucm:aggregates ?component)
    (?component ucm:implementsCapability ?capability)
    (?component rdf:type ucm:Component)]
```

This rule depends on the fact that *implements*, *fires* and *hasContext* are sub-properties of *implementsCapability*.

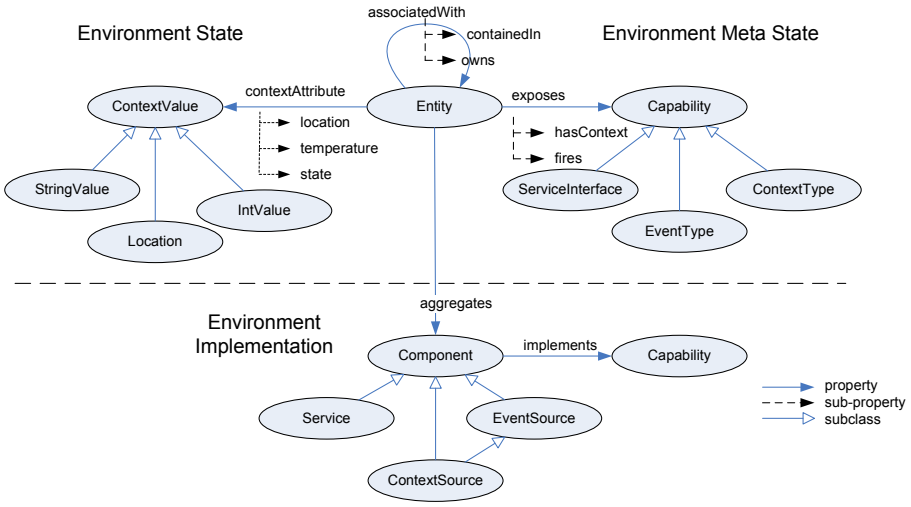


Figure 3.7: Key objects and relationships of the UCM.

3.3.4 Summary

The three aspects of the Environment model and how they relate to one other through an entity is summarized in Figure 3.7. The State, Meta-State and Implementation aspects are all related to an entity instance. Furthermore, entities may be related to each other by entity relationships.

3.3.5 Model Example

An simple example of the model is illustrated in Figure 3.8. A person entity *personBob* is shown to hold a mobile phone *bobsMobilePhone*. In the current Environment State, this phone has a location called *phonePosition* containing the longitude and latitude of the device. In the current Meta-State, *bobsMobilePhone*

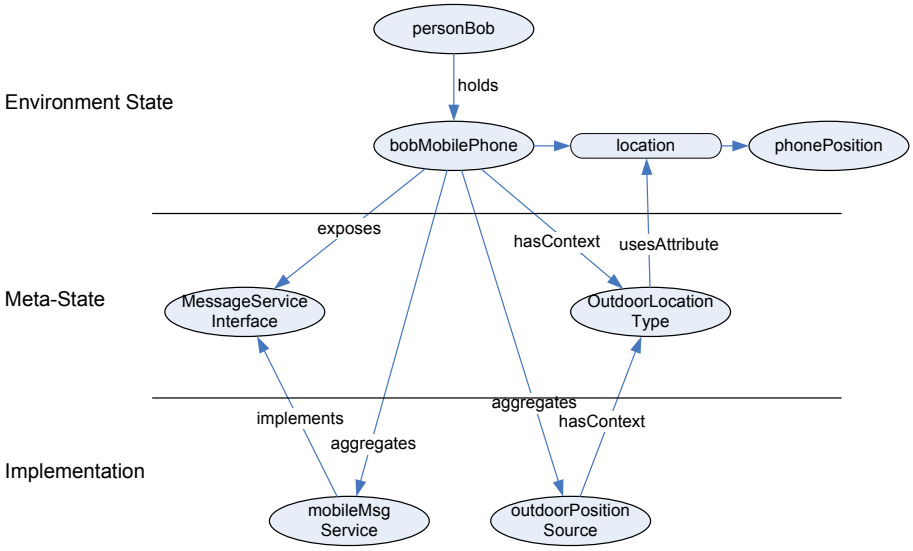


Figure 3.8: Model example illustrating State, Meta-State and Implementation aspects.

is shown to expose a *MessageServiceInterface* and has the *OutdoorLocationType* of context available for applications. The *OutdoorLocationType* uses the *location* attribute as shown. In the current Implementation, *bobsMobilePhone* enitity aggregates the *mobileMsgService Service* component and the *outdoorPositionSource ContextSource* component. The *mobileMsgService* implements the *MessageServiceInterface* type, and the *outdoorPositionSource hasContext OutdoorLocationType* as shown. The component aggregation rule ensures that *bobsMobilePhone* exposes the same *MessageServiceInterface ServiceInterface* and *OutdoorLocationType ContextType* to applications.

In this example, the *mobileMsgService* and *outDoorPositionSource* implementation components would be supplied by integrated systems. For example, a Context Toolkit Widget could be mapped to the *outdoorPositionSource* component of the UCM, while an SMS service could be mapped to the *mobileMsgService*.

3.4 Model Discussion

In this section, we discuss how the UCM addresses the requirements outlined in Section 3.1, specifically how it supports application portability, environment specialization, introspection and mapping to existing systems.

3.4.1 Portability

When a system is adapted to the core UCM as described, some degree of interoperability is possible. Applications can browse an environment by entity relationships, and display the types of context and services associated with these entities for example. Applications can identify entities, context, services and events. However, a higher degree of interoperability is only possible only when applications share a deeper semantic understanding of entities and their associated resources with the supporting infrastructure. To address this we propose the use of *Environment Profiles* described next.

3.4.2 Specialization

To support specialization, *Environment Profiles* specialize the core model for specific environment types. Profiles will contain the specific classes of entities, services, context, events, content and their possible relationships for a given environment type. A home profile, for example, can consist of typical place entities in the home such as kitchens and living rooms, device types such as appliances, and home entertainment systems. Home context and services can include temperature, lighting controls and room-resolution location sensors. An application interacting with the home environment can then “turn the lights on in a room” when a user arrives by specifying the expected lighting service associated with a room. Similarly a museum profile could define displays, visitors, galleries, display content,

visitor location, and interests. Profiles may be extended further by an integrator to provide extensions specific to a deployment, at the possible expense of interoperability. Through the use of a specialized core model, and supporting infrastructure to map this model to existing systems we argue that it is possible for applications hosted outside of an environments local domain to interoperate with an environments resources independent of the ubicomp middleware used.

3.4.3 Introspection

The Meta-State aspect of the model supports introspection for applications to determine whether its requirements can be met by the environment. An entity can be queried for the context types, service interfaces and event types it currently supports. The environment as a whole can be queried for the entities that match a given criterion such as current context values, entity types, and the types of context, events and services exposed.

3.4.4 Mapping to existing systems abstractions

Since the UCM design is based on a thorough analysis of existing systems, our model *should* lend itself to a straightforward mapping to a subset of these systems. The separation of the Implementation aspect from the exposed State and Meta State lends itself to a straightforward mapping assuming there is a correspondence between the component types of the UCM Implementation aspect and those of an integrated system.

For example, the *ContextSource* is similar in concept to a Context Widget in the Context Toolkit, providing the capability to query and subscribe to context changes. The *EventSource* component can be used to describe the iROS Event Heap, providing the capability to subsribe to and receive arbitrary events. UCM

Services can be used to describe service oriented systems interfaces. When a system such as Cooltown, or Parc TAB provides servers that aggregate context and services around entities, the *EntityHandler* can be used to describe these components in the UCM. Since an instance of the UCM describes an explicit environment model, adapters for smart space systems such as Sentient Computing and EasyLiving can keep their model in sync with the UCM by adding and removing static and dynamic entities and relationships as they change.

Furthermore, the separation of Implementation concerns allows a supporting system to vary component aggregations independently of the exposed capabilities. This allows more than one system component to provide capabilities to a single entity, or many entities to make use of a single component in an underlying system. An implementation of the UCM may change entity-component aggregations depending on the situation.

The key to integration is the use of *adapters* which will supply an instance of the UCM with descriptions of the components and entities of the integrated system as they are added and removed. The adapter will also delegate method calls from UCM applications to the integrated system's as appropriate by maintaining a mapping of UCM component descriptions to the underlying system capabilities. The use of adapters for integration is discussed further in Chapter 4.

Assuming the Implementation aspect is used to map existing systems abstractions to a common model, at best we should expect to mirror the "native" environment programming model presented by a given system. In some cases, however, we may need to present the UCM as an *alternative* programming model, one that is either higher level or lower level than the native model. The tradeoffs in providing the UCM as an alternative programming model for existing systems is explored further in Chapter 5 by creating applications that make use of several concurrently integrated ubicomp systems.

3.5 Access Control and Security

As ubiquitous computing systems become more widely deployed, the need to protect access to computing resources such as sensors and actuators becomes more necessary. Sensors can record movement, activities and other information about users in areas that can threaten user's privacy. With the use of a common model for ubicomp we can anticipate the need to provide access control for previously assumed private or closed ubicomp deployments.

Recent research in the areas of security and privacy have explored the use of new metaphors for ubicomp security (e.g. virtual walls [67]) and the use of lightweight authentication approaches for resource constrained devices such as that used in Bluetooth [86] and the Unmanaged Internet Architecture (UIA) [41].

To highlight the UCM's extensibility, we describe how the UCM (Environment Meta State) could be extended to support an *access control* abstraction used to represent both security domain and access control mechanism. We then provide an example of how this model extension could be supported in a future integration platform. We chose the Unmanaged Internet Architecture [41] in our example since it uses a novel peer to peer authorization mechanism, includes device group management and can serve as a secure base communications platform for other ubicomp systems (e.g. MyNet [66]).

In the UCM, an application accesses the *capabilities* of an an associated entity. Note that we use the term *capabilities* as introduced earlier in this chapter, in the UCM sense, recognizing that this is not the same as *capabilities* in a capability-based operating system[1]. Recall that in the UCM, capabilities are the super class of ServiceInterfaces, ContextTypes, and EventType as shown in Figure 3.7. For

[1]In capability-based systems capabilities are defined as unforgeable references to objects that allow access to well defined operations on operating system objects such as files and devices.

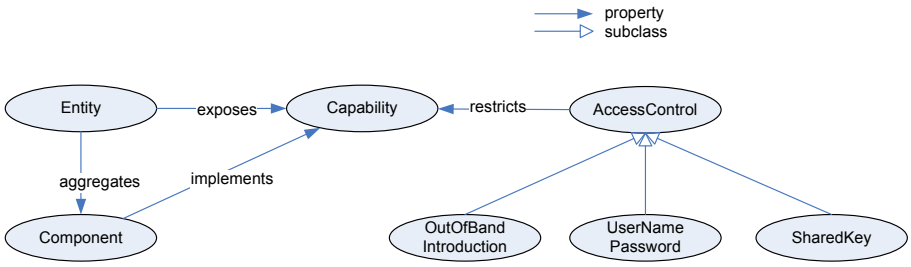


Figure 3.9: Access control property associated with capabilities.

example, in the home there may be a UCM capability (ServiceInterface) that allows an application to control the lights in a room (entity). While UCM capabilities are also references to computing resources, they are not unforgeable, and represent only the *potential* use of an integrated computing resource: access is by no means guaranteed. In the remainder of this section we use the term *capability* to refer to a *UCM capability* and not a security capability.

In a UCM deployment, computing resources in a ubicomp system are typically provided by one or more underlying systems' adapters by adding UCM *components* to the model. UCM Components *implement* capabilities. When a component is *aggregated* by one or more entities, these entities *expose* these capabilities as shown in Figure 3.9. UCM applications access computing resources by interacting with entity capabilities.

Here we assume that the computing resources accessible by an application are controlled as a group in a single administrative, network domain, perhaps within virtual walls [67], or individually (e.g. using Bluetooth). We can expect that the credentials required for an application to access resources in a ubicomp system can vary from something as simple as a PIN, a user-readable string of words (used for out of band authentication [41]), or a public key.

To illustrate how the core UCM can be extended to model both security do-

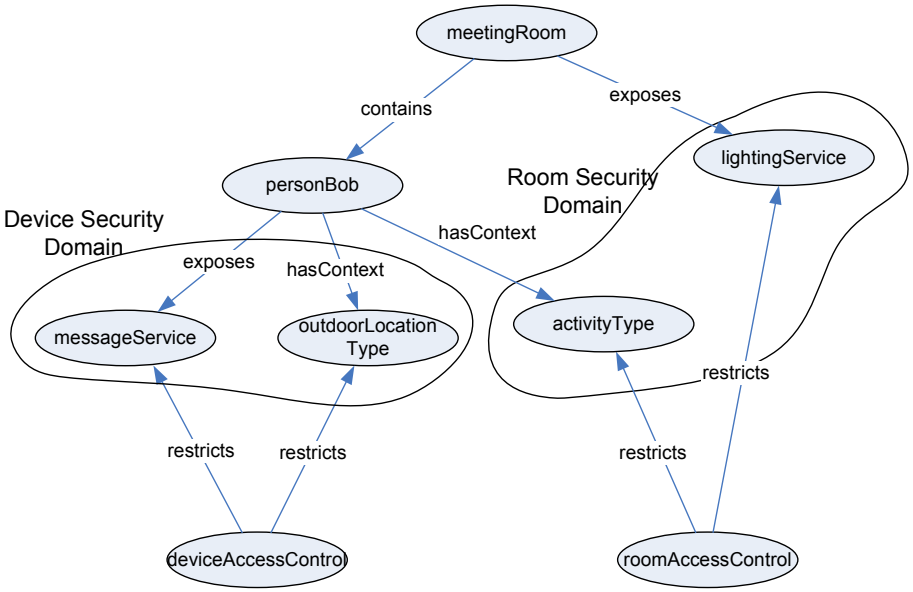


Figure 3.10: Example of AC properties used to mark security domains.

mains and access control required to interact with computing resources in a ubicomp system, we can add a new core abstraction we call an *access control* (AC). We say that an AC *restricts* capabilities as shown in Figure 3.9. When a capability is *restricted* with an AC property, this indicates that the capability is a member of a security domain (such as a LAN, a server or individual device like a laptop) that may require participation by the UCM application in an access control mechanism (e.g. an out of band introduction, PIN, public key). In this way, entity capabilities in the UCM are grouped by domain and access control type as illustrated in Figure 3.10.

In the model example illustrated in Figure 3.10, the entity personBob has two capabilities, one (a ContextType) is used to retrieve Bob's location (outdoorLocationType), another (a ServiceInterface) allows applications to send SMS messages to Bob (messageService). Another capability allows us to retrieve Bob's current

activity (activityType). A ServiceInterface of the room entity (meetingRoom) allows applications to control the lighting (lightingService). Access to the outdoorLocation and messageService are controlled by Bob's device, whereas the activitySource and lightingService are controlled by a ubicomp system installed in the meeting room. To indicate this, the messageService and outdoorLocation are restricted by an AC instance called *deviceAccessControl*; the activityType and lightingService are restricted by the *roomAccessControl* as shown. Note that there is not necessarily a correspondence between how entities group capabilities and security domains marked by an AC instance.

By associating access control instances with capabilities, and subclassing a core AC abstraction class as shown in Figure 3.9 we can extend the UCM to include information about both the security domains and access control mechanisms used in integrated systems.

Using this extension to the UCM it should be possible to limit the access and visibility of specific integrated resources to UCM applications. For example, each UCM application may have an associated access control list. When the model is queried in an integration platform like the UIF described in Chapter 4, this list can be checked against the capability AC restriction to either hide or expose the capability for that application. The use of AC abstractions in an executable model may also be used by a supporting system to reason about security domains at run time, adjusting security domains depending on the context of certain entities for example.

Note that neither capabilities nor their associated AC identifiers are used as the credentials needed to access a system's resources. ACs represent a security domain and the mechanism required in the UCM. We expect to rely on the underlying system to ensure that a UCM intermediary authenticated and secured within their security domain(s).

3.5.1 Security Example

As an example of the use of the access control abstraction, we discuss how the Unmanaged Internet Architecture (UIA) [41] security domain and access control mechanisms could be supported by a UCM implementation. This may be necessary when the user wishes to access personal devices such as a mobile phone from a UCM application.

The UIA is a peer-to-peer connectivity architecture where each user is the administrator of his or her *personal group* containing their mobile and personal devices. Users can "merge multiple UIA devices to form a personal group, after which the devices work together to offer secure access to any device in the group from any other."[41]. Similarly, users can create *shared groups* to share access of their personal devices with others. When a user wants to add a new device to a group, the UIA finds other devices in the group (e.g. one discovered on a wireless LAN or via social network). The user then selects "Introduce Devices" on the new device and one already in the group to start an *introduction* process. An *introduction key* consisting of three words chosen randomly from a dictionary is shown on the display of both devices. Users then choose the other device's introduction key from a list of three other random keys to complete the introduction. If a matching key is not found, the procedure is aborted. The user ensures that the introduction key of the other device is correct since it is highly unlikely that an impersonator on the wireless LAN will supply another random key that matches. Other aspects of the UIA security mechanisms are discussed in [41]

In a typical UCM deployment, UCM applications will be proxied by an intermediary such as the UIF platform and adapter described in Chapters 4 and 5. To access devices in a UIA personal group, an adapter must become a member of that group using the UIA introduction process as shown in Figure 3.11.

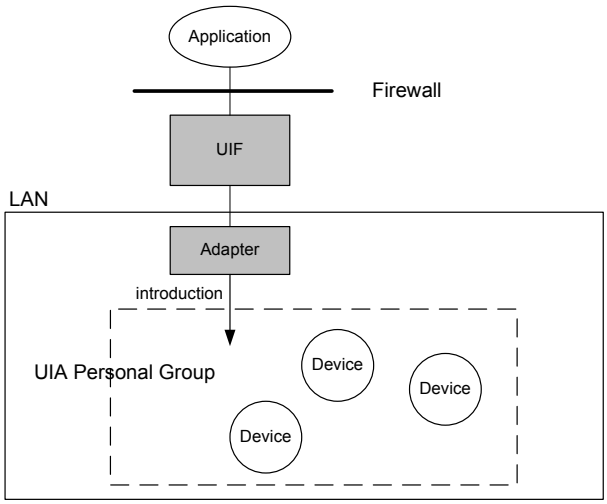


Figure 3.11: Access control and security example: introducing a UIF adapter into a UIA personal group.

To do this, capabilities discovered in the LAN by an integration system adapter (as described in Chapter 4) are marked (i.e. *restricted*) with an AC instance that represents a UIA group. The type of the AC instance is used to indicate that an "out of band" introduction process is required. An example of this is shown in Figure 3.12. Bob has two capabilities supplied by his mobile phone restricted by the uiaPersonalGroup. The uiaPersonalGroup is an instance of an OutOfBandIntroduction AC abstraction as shown.

PlaceMedia is a web application that displays the positions of friends and places of interest on a Google map [40] as described in Section 5.1.1. For a UCM application such as PlaceMedia to interact with a capability, the integration system (UIF) delegates calls to an adapter as described in Chapter 4. The adapter then processes the call by marshalling the request and response to and from the integrated system.

For example, the PlaceMedia application would like to retrieve the position of

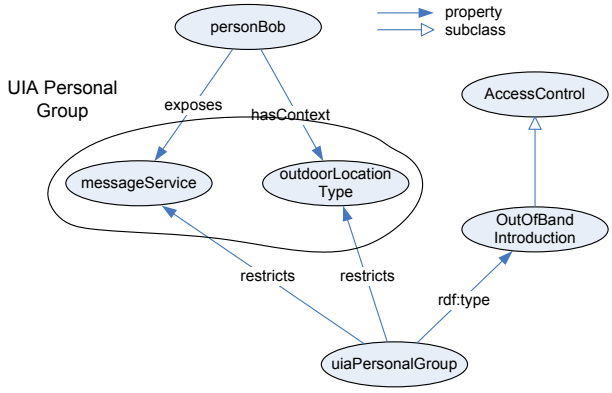


Figure 3.12: Example of capabilities restricted by UIA personal group.

the user to display on a map. This could be done by accessing a user's mobile phone which is a member of the user's UIA personal group. To do this, the UIF delegates the location (context) request to the UIA adapter. The UIA adapter can then delegate the call to a UIA device to process the request.

When the adapter is not yet a member of the UIA personal group the phone belongs to, the UIA adapter will need to be introduced to the device. To do this, the adapter can trigger the UIA introduction process and return a "challenge" response to the UIF. The response will include public information needed by a PlaceMedia user to complete the introduction. The UIA *introduction key* of the adapter can be included in the challenge.

If the adapter host is near the mobile phone, the standard UIA introduction user interface can be used on both the adapter and the phone to complete the adapter-phone introduction.

For situations when the PlaceMedia user is not near the mobile phone, PlaceMedia can display the adapter's introduction key supplied with the challenge response. The introduction between the adapter and the mobile device can then be

completed remotely by the PlaceMedia user and the mobile phone owner by voice, SMS or email to ensure the introduction keys match [41]. When both users determines that the intoduction key is correct, PlaceMedia can continue the process by making an "authentication" call with the remote introduction key to signal the adapter to complete its side of the introduction process. The mobile phone user can complete his or her side using the standard UIA interface.

Once authenticated, future requests to the mobile phone in the UIA personal group will now succeed since the adapter is now an introduced member of the UIA personal group. As described in [41], the owner of the personal group can revoke the adapter's membership at any time; the adapter will need to be introduced again to access the mobile phone.

To support other authentication mechanisms, other (public) information can be provided in an authentication challenge and/or response in the same way. For example, public keys carried with an end user device such as a mobile phone as in the Instant Matchmaker [93] can provide the necessary credentials when a UCM application is challenged for access to a resource. Of course, integration with the integrated systems' access control mechanism assumes there is an API, or code is available[2] to allow an adapter to trigger the introduction process and retrieve the introduction key displayed in the dialogs as described.

Our treatment of this example is deliberately informal and is meant only to illustrate how a given security framework like the UIA may be supported by the UCM access control extensions and an integration system. We recognize that given a specific security framework, an implementation may require a thorough and formal analysis to ensure all security requirements are satisfied. However, from our initial investigation and as described above, we are optimistic that the UCM is able to gracefully support a meta level security model that maps naturally to underlying

[2] see http://pdos.csail.mit.edu/uia/

security models as implemented by existing systems.

3.6 Use Cases for an Executable UCM

To use the UCM for creating and maintaining an integrated ubicomp environment, it must be instantiated in a supporting sytem. By expressing the UCM explicitly using semantic web languages such as the Web Ontology Language (OWL) [102] and the Resource Definition Framework (RDF) [101] it is possible for such a system to store, query and infer new information about an integrated ubicomp environment using Semantic Web tools and libraries such as the Jena Semantic Web Framework [1] in an integrated system. In this section we summarize several environment design/integration time (static), and run time use cases for for an *executable* UCM.

3.6.1 Design/Integration Time Use Cases

At environment design time, the UCM expressed as an ontology can be used to describe the initial state and static aspects of the environment model. This includes the following:

Static environment model descriptions. In any environment there are static entities such as rooms, furniture, and fixed devices. The UCM can be used to describe these entities and their static context such as the location of fixed objects or the name and email address of a user. Important static relationships such as relative location can also be described. For example, a printer is *containedIn* a room, and a meeting room is *nextTo* an office.

Interface semantics and context quality. Even when a component exposes the same interface, the behavior or quality of its service may vary from one implementation to another. A GPS location sensor may provide location accuracy of 5m outdoors whereas an indoor location sensor may provide only room level accuracy.

Semantic descriptions can make the quality and semantics of an interface explicit to applications so that they know when they can and should be applied.

Component configuration information. In a supporting system such as the UIF described in the next chapter, components in the system may be implemented by native Java classes or by integrated ubicomp middleware. In the UCM, a *binding* property of a component is used to associate a component in the model with a native Java class implementation, while an *adapter* property associates the component with integrated middleware.

Initial entity-component dependencies. A physical entity such as a device will often have services, context or events associated with it. The aggregation relationship between entities and implementation components establishes the initial link between the conceptual or exposed model of an environment and its implementation components.

3.6.2 Run Time Use Cases

A challenge for any ubicomp system is in managing the dynamic aspects of an environment for applications. In our system, the UCM used with an associated reasoning engine can be used to maintain the exposed model and associated components as their composition changes. We have explored the following run time use cases for an executable UCM:

Entity Composition. When new entities such as mobile users and their environment arrive and leave the environment, associated information about these entities can be added or removed from the model. A reasoning engine can then associate these new entities with the appropriate components, types and service interfaces.

Context Inference. Ontology-based systems such as [27, 51] have shown that

it is possible to use an integrated reasoning engine to infer higher level context from lower level context information. Cooltown [68] and other systems have demonstrated the value of entity relationships for service discovery and other use cases. A reasoning engine with appropriate rules can be used to infer such relationships from lower level context information. For example, the following rule establishes the near relationship when one entity is within 50 metres of another.

```
[(?entity1 ucm:near ?entity2) <-
    (?entity1 ucm:location ?position1)
    (?entity2 ucm:location ?position2)
    notEqual(?entity1, ?entity2)
    inRange(?position1, ?position2, 50)]
```

Dynamic component call/message dispatch. At run time, an environment model described using the UCM can be queried to determine the component that currently supplies the requested service or context which may depend on the current situation. For example, one context source may be used to determine a user's location indoors and another outdoors. The following rule will aggregate a context source (*csIndoorPositionSource*) for providing indoor position information only when a Person entity is *containedIn* the *csBuilding* place.

```
[(?entity ucm:aggregates campus:csIndoorPositionSource) <-
    (?entity rdf:type ucm:Person)
    (?entity ucm:containedIn campus:csBuilding)]
```

Entity classification and discovery. A client of the system or internal component may need to find the entities in the environment that match a certain criteria such as its type, the service interfaces, or context types they provide. Similarly, clients or components may need to know the interfaces or context supported by a given entity.

Situation events. With an integrated model of an environment, it is possible to query or be notified when a certain situation exists in the environment as a whole.

For example, one can query whether there is more than one user entity contained in a meeting room. This may be used by an application to infer that a meeting may be starting to call appropriate services.

Security. If the model is executable, the capabilities accessible (e.g. associated with access control properties as described in Section 3.5) by UCM applications can change depending on the context of various entities such as users location in the model, permitting context-based security mechanisms. Of course there are a number of assumptions such as the integrity of context that are critical for consideration in the use of the UCM in these scenarios.

3.7 Summary

In this chapter we presented the requirements for a common programming model for ubicomp based on core abstractions shared by several representative ubicomp systems surveyed in Chapter 2. These shared abstractions are an *environment model*, *entities*, *entity relationships*, *context*, *services*, *events* and *data* or *content*. Using a simple example we demonstrated how these abstractions can be used to model an environment leading to the the design of the Ubicomp Common Model. The UCM is an entity-centric model consisting of three related aspects: the Environment State containing context and entity relationships, Meta-State containing entity capabilities such as context types and service interfaces, and Implementation containing implementation components such as context sources and services. We discussed the UCM's suitability for meeting several requirements of a common model: application portability, specialization for different environments, introspection, its suitability for adaptation to existing systems, and extensibility to support emerging security mechanisms. Finally we presented use cases for an *executable* UCM in a supporting system such as the Ubicomp Integration Framework (UIF).

The UIF used to evaluate the UCM's suitability for ubicomp systems integration is described next.

Chapter 4

The Ubicomp Integration Framework

In this chapter we present the Ubicomp Integration Framework (UIF), a *meta*-middleware platform that uses the UCM for both application development and ubicomp system integration. Its primary use is to expose the functionality of *existing* ubicomp systems in a controlled and unified manner. In this way the UIF can then be considered a sophisticated *wrapper* or *Façade* [44] that encapsulates one or more ubicomp system with a service interface. In reality the UIF is more complex than its role implies since it provides a composite model of an integrated environment, and so contains much of the functionality of other ubicomp middleware platforms. In this chapter we describe the architecture and implementation of the UIF system and associated adapter framework in some detail. Readers who are more interested in the use of the UCM for application and adapter development than the UIF implementation details may prefer to skip much of this chapter and instead turn to the summary in Section 4.9.

4.1 Analysis and Approach

The UIF is used to assess the feasibility of using the UCM to unify the programming model of existing ubicomp systems for both application portability and sys-

tems interoperability. In the next Chapter (5), we describe our experience using the UCM in the UIF to unify the abstractions of four representative systems to manage a *composite* environment model. This composite model can then be accessed by application developers using a single API.

To accomplish this, we require software to map the various data formats, interfaces and protocols of underlying systems to a common protocol, effectively presenting a homogeneous view of the integrated underlying ubicomp systems to the integration framework. Unlike other ubicomp systems that also provide a homogenous execution environment for applications such as those in discussed in Chapter 2 and others (e.g. [72] and more recently [14]), we aim to accomplish the same goal by integrating *existing* ubicomp systems, accessing as much of their functionality using their existing APIs. This will demonstrate the feasibility of using the UCM to integrate systems and how well the UCM is able to capture the abstractions of representative systems.

4.1.1 Analogy to Enterprise Application Integration

In the Enterprise Application Domain (EAI), integrators face similar problems when attempting to integrate diverse applications in a business. Unlike the EAI domain however, our main focus is not to integrate applications, but middleware systems that coordinate computing resources. Because of this, several well understood (and often less arduous approaches) to enterprise integration cannot be applied directly.

For example, in some cases, enterprise applications are integrated at the *presentation* or user interface level [71, 88]. This can be done by screen scraping the integrated application, or combining the user interfaces into a web portal. The advantage of this approach for the integrator is that it is relatively easy to accomplish,

but it often means that there is strong coupling between the integrated application or interface and the integrated system. This approach is not feasible for our task since several ubicomp systems we aim to integrate do not always have a presentation layer or user interface. Another enterprise integration approach is to integrate the *data* held by applications. This may involve data replication, federating data, or the use of interfaces that provide access to the data in integrated applications. This approach is sub optimal for our integration task since we need to access not only the changing environment model and state of an integrated system, but also the functionality of those systems.

Finally, enterprise integrators have used a *functional* integration model that allow potentially new applications to invoke existing functionality from new applications [88]. This is done using the available APIs to the integrated applications. To ensure we are leveraging the programming abstractions provided by an integrated ubicomp system through its APIs, and to validate our integration model, this is the approach (necessarily) taken in our integration task for evaluating the UCM.

Within the functional integration model, an integration can ensure data consistency across applications, coordinate actions across integrated applications, and use well defined service interfaces (called a service oriented [71] or plug and play integration [88]). In our task, we similar goals in that we aim to ensure that aspects of the composite environment model held in an integrated system are kept in sync with that of the integrated ubicomp system's model (data consistency). We also aim to use existing systems in a coordinated manner to add new functionality (e.g. applications, composite services, inferred context and entity relationships) in the integration framework itself. Enterprise integrators typically use *adapters* for wrapping applications in a well defined interface [12, 71, 88]. A different adapter is typically required for each integrated application. For integrating ubicomp systems, we take a similar approach, providing an adapter for each type of system we

aim to integrate. In our case we use the same interface for all adapters.

Although not all EAI approaches can be directly applied to ubicomp systems integration, we were influenced by well known EAI methodologies and more recent enterprise middleware implementations using ontologies (e.g. [78]). An important step during analysis for EAI is to determine the high level "business entities" in an enterprise leading to the identification of the component interfaces needed for integration [88]. In the EAI domain the basis for identifying entities is often the various functions in an enterprise such as R&D, Production, Marketing, Sales, Distribution, Service, Accounting, and Personnel. In EAI as in ubicomp, there are also cross functional entities like *people* and *organizations*. In the ubicomp integration domain, our basis for identifying entities and components are the abstractions present in representative ubicomp systems (e.g. context, events, services, content) and the physical world (people, places, things, their relationships) as described in Chapters 2 and 3. Unlike typical enterprise applications, ubicomp systems are generally cross functional in the sense that they are (typically) designed to support a variety of applications.

In both the EAI and ubicomp domains, the semantics of the various abstractions must be consistent for cohesion in a unified programming model for new application developers. Because of this, many working in the EAI domain have begun to use ontologies "to provide a shared and common understanding of data (and in some cases, services and processes) that exists within an application integration problem domain"[1]. While recognizing that there are differences between the abstractions of an enterprise integration and ubicomp, we also employ ontologies. In a typical EAI deployment, however, the available functionality and its relationship to business entities is often fixed at design time. In an integrated ubicomp environment, the available functionality and entities will change dynamically

[1] [71] pp 394

as discussed in Section 3.6.2. Because of this, we need facilities to for managing a integration model that can be changed at runtime by adapters, applications and by the integration framework itself.

4.1.2 Environment Model Management

Since our goal is to unify these systems in a *composite* environment model, we also need a way to manage this model (i.e., an instance of the UCM). The UIF model subsystem will provide the rest of the integration framework with the information needed to correctly select and make use of an underlying system to satisfy a given application request.

In a composite ubicomp environment, multiple integrated ubicomp systems will typically be involved in context provisioning or in the implementation of integrated services. The underlying services and sensors used by an application at a given time may also depend on the situation. For example, in an indoor environment, location information may be provided by a Context Toolkit widget, but when the user is outdoors, location may be provided by wide area mobile phone infrastructure. We need an approach that allows an integrator to coordinate both the static and dynamic aspects of an integrated environment model. More specifically, we need an approach that provides the integration framework with the knowledge of the static and dynamic relationships between computing resources such as software services and sensors, and people, places and things.

We can satisfy this requirement by instantiating the UCM directly in the integration system itself. This will also satisfy a requirement for our model highlighted in Section 3.1, that is, for the model to be *executable*, i.e., queried and reasoned with by the system. Applications will query the model to discover entities and their capabilities for interaction and to determine whether their requirements can be sat-

isfied. Direct support for reasoning in the integration platform will also allow the system to satisfy the run time use cases presented in Section 3.6.2 such as entity composition, context interpretation or inference, and dynamic call dispatch.

Initially we considered meeting these objectives using a relational database such as MySQL [8] or another data store. After some initial prototype work however, we decided to make use of semantic web technologies to manage our integration model. This decision was based on four factors.

1. We first considered the **flexibility** that the use of semantic web technologies offers in terms of modeling, query (e.g. using SPARQL [103]) and reasoning support, gaining an understanding through prototypes and experimentation.

2. We noted the **increasing breadth and depth of tools available** to create, and manage ontologies and models that use OWL and RDF. Design tools such as the Protg ontology editor [9] and plug ins for the popular Eclipse IDE [4, 6] are available for ontology design. Frameworks and toolkits [24, 73], high performance knowledge bases [53, 110] and reasoning engines [52, 92] are also available.

3. Since we aimed to integrate existing ubiquitous systems using application servers we also considered **recent enterprise middleware approaches** called *Ontology Driven Architectures* (ODA) [78, 100]. ODAs use an integrated knowledge base and reasoning to ease the development and management of applications hosted by enterprise application servers.

4. The use of Semantic web technologies has emerged as **a well established approach to managing context and configurations in ubiquitous systems** as demonstrated by Gaia [84], the Context Broker Architecture (CoBrA) [26–28], the Service Oriented Context Aware Middleware (SoCAM) [51] and

others [30, 99].

Based on these considerations, we decided to make use of semantic web technologies and an Ontology Driven Architecture to assist in the configuration and run time maintenance of the composite environment model managed by the UIF. An instance of the Ubicomp Common Model (UCM) is created by an integrator using to specify the static configuration of the environment. The desired response to dynamic run-time changes such as the addition of new entities and components are described using rules executed by an integrated reasoning engine [1] that maintains both the exposed aspects of the model (Environment State and Meta State) and the implementation as the composition of the environment changes.

4.1.3 Cross-Domain Interaction

Since we aim to support access to ubicomp systems from applications outside of the administrative domain where an integrated ubicomp system is deployed, we provide a standards-based Web Service interface to the UIF. The use of Web Services makes the UIF accessible and potentially discoverable on the Web through firewalls and across wide area networks. With a standards-based wide area security framework [81] Web Services can be accessed in a secure manner. Other mobile and ubiquitous systems such as ReMMoC [45] and WSAMI [62] as well as commercial systems such as UPnP [74] have proven the value of providing a Web Service abstraction around individual environment resources. While these systems have demonstrated that the use of Web Service is a workable solution in some cases, our aim is more ambitious: unlike these systems, we aim to wrap *whole systems* that expose higher level abstractions such as *environment models*, *entities* and *relationships*, as discussed in Chapter 2, not just individual components or services.

While the use of other protocols is possible, there are several benefits to the use of Web Services [109]. They are relatively simple to use with application servers, especially with the variety of tools and frameworks available for virtually all programming languages today. Based on industry standards, they have been widely adopted for cross-domain communications. Web Services promote loose coupling, since only the service and connection is described, independent of the implementation at either end. They are self describing and discoverable on the Internet. Finally, while HTTP is often the transport used, other transports can be supported by a given Web Service for more efficient internal communications over private networks. However, despite the numerous benefits there are some drawbacks to the use of Web Services, particularly in the area of performance. Scenarios like ours will incur overhead since the Web Service tools transform method calls to and from XML-based SOAP messages. Furthermore, our integration system will necessarily add an additional layer on existing systems for routing method calls. However, through the careful application of Web Services best practices [31], we believe that the benefits of using Web Services outweigh these cost. In the following sections we describe the implementation of the UIF including our use of Web Services in more detail.

4.2 Implementation Overview

The UIF is a tiered enterprise server application as shown in Figure 4.1. The system performs three essential functions:

1. It serves as a repository for knowledge about a composite environment model. This "knowledge" consists of information contributed directly by an integrator, by ubicomp system adapters at run time, or deduced by an integrated reasoning engine and integration rules.

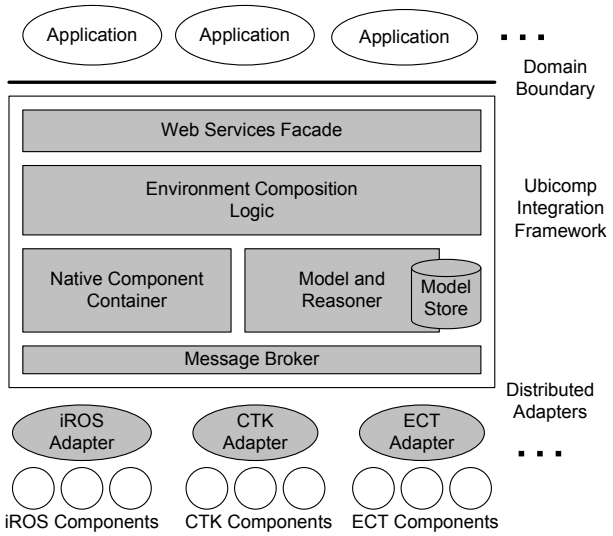


Figure 4.1: Ubicomp Integration Framework Architecture.

2. The UIF dispatches method calls from applications to the appropriate (distributed) adapter or internal components based on the raw and inferred data in this repository.

3. The system manages event subscriptions for clients of the composite model, ensuring subscriptions to asynchronous events are propagated correctly and maintained as long as the corresponding integration adapter is available.

Figure 4.2 illustrates the typical interactions between UIF subsystems when an application requests context for an entity. First, the application makes a call to the Façade Web Service to get context (1). This is delegated to the Environment Composition Logic (ECL) (2) which makes a request to the Model and Reasoner subsystem (3) to get the component information that handles the specified entity and context attribute (4). Based on this information, the ECL then queries a native component hosted by the Native Component Container for the context value (A.5)

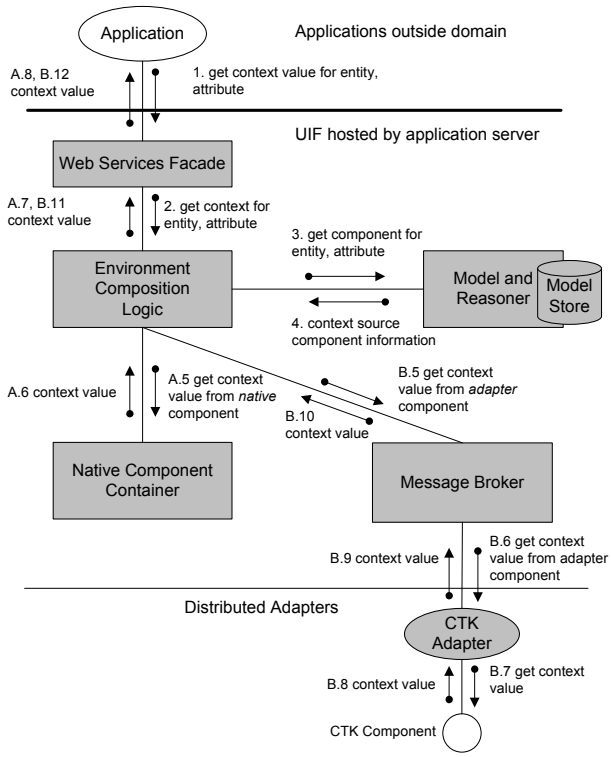


Figure 4.2: High level interaction between UIF subsystems.

or forwards it to the Message Broker (B.5). If a native component handles the request, the context value is returned to the application (A.6, 7, 8). If it is handled by an integrated system, the Message Broker forwards the request to a distributed adapter (B.6), in this case the Context Toolkit (CTK) adapter. The adapter then makes the query to its component (B.7) and the context value supplied is returned to the application (B.8, 9, 10, 11, 12). Note that it is also possible for (static) context to be maintained in the model. In this case, the ECL will query the model for the context value (not shown).

The Façade subsystem provides a SOAP-based [104] Web Service interface to applications for cross domain interoperability. As described, calls to the Façade are

delegated to the Environment Composition Logic (ECL) subsystem. The ECL is not only responsible for dispatching queries, but also for maintaining subscriptions to asynchronous events.

The Model and Reasoner subsystem maintains the current environment model including the UCM itself, specializations of the UCM, entity instances, static context values, capabilities, component descriptions and their relationships. Queries for entities based on their types, capabilities and static context are handled directly by the Model subsystem since this information is maintained in its knowledge base. Based on rules supplied with the UCM and an integrator, an integrated reasoning engine can establish new relationships depending on entity types and context values when components or entities are added or removed from the model by an adapter or an application.

The Native Component Container hosts internal "native" UCM components instantiated by the system either on start up, or when first accessed by an application. A typical use of a native component by an integrator is to provide a composite service, or a specialized context inference capability for the integrated environment. For example, in the case where user entities aggregate an SMS messaging service, an integrator can create a new service to send SMS messages to friends of a user, or users contained in a certain building or room. This new broadcast service makes use of the SMS messaging services of an integrated system. Similarly, an integrator could create a simple context source that provides the number of users in a room by counting the entities contained in that space.

Method calls destined for an adapter are dispatched by the Message Broker as described. Distributed adapters transform the method call to and from the integrated system's data structures and APIs as needed.

To support asynchronous events, applications supply subscription parameters specific to the event type. The adapter associated forwards the subscription to the

native system and maintains an internal mapping to the underlying system subscription. Later, when an event is signalled by the integrated ubicomp system, its adapter marshals the event data to a common UIF `EntityEvent` data structure, and sends it to the UIF. The ECL looks up the associated subscriber and queues the event for application retrieval.

The UIF was implemented using the JBoss [7] Java 2 Enterprise Edition (J2EE) [97, 98] server, a fairly standard platform for enterprise application development. The UIF consists of approximately 7800 lines of Java code and other components. The UIF Model subsystem wraps a Resource Description Framework (RDF) store and general purpose rule-based reasoning engine supplied with the Jena Semantic Web Framework [1]. The initial model of the environment consists of the UCM and environment-specific ontology loaded when the system starts. Static entity, context and component descriptions are also loaded at start up, along with default rules supplied by an integrator. Our prototype deployment described in Chapter 5 consisted of 535 RDF triples and 278 rules. The Broker communicates with Adapters using Remote Method Invocation (RMI) [38] so they can be distributed in an integrated environment. The adapter framework and implementations are approximately another 3000 lines of Java code. In the following subsections we describe each subsystem in more detail followed by a discussion of our adapter framework.

4.3 Façade

To implement the UIF Façade Web Service we have used both the Axis [10] and JBossWS [5] Web Service frameworks. These frameworks can provide server and client side code to marshal method calls to and from SOAP, and generate WSDL from an supplied Java class or interface. Using tools supplied with either of these

frameworks we generated the appropriate server side deployment files and WSDL for the Façade interface.

One challenge specific to our integration task, however, is the need for a generic data type that is flexible enough to support complex service parameters, event data and context information while making it easy to marshal to SOAP by different Web Services frameworks, not only for Java applications but other languages such as C# and C++. While we could have considered the use of RDF as a data exchange format, we did not want to force application developers to include an RDF parser. We required a data structure that can be marshaled easily to and from the various data formats needed by an integrated system, and to and from the RDF-based knowledge base and reasoning system in the UIF. To accomplish this we created the `DataObject` data structure and associated utility classes for converting any data structure to and from native Java and RDF. Fields in a `DataObject` are strings (names, values), integers (type) or nested DataObjects to support complex data structures and arrays; it should be a straightforward task to provide `DataObject` converters for other languages. Applications typically exchange data with the Façade Web Service using `ContextValue` and `EventSubscription` data structures that use the `DataObject` where a generic data structure is needed.

There are four categories of Façade methods:

- **Application authentication**. The system must be able to authenticate applications that are allowed to make use of the integrated environment. In the UIF, an application must log in with its client id before it can access the model. The following methods of the Façade interface are in this category.

```
// Application authentication
public String login(String clientId,
  String password) throws RemoteException;
public void logOut(String clientId) throws RemoteException;
```

- **Model Composition**. Methods in this category allow an application to add and remove entities such as users, objects, places and other entities to the model. These include the following methods:

```
// Model composition
public String addEntity(String clientId,
  EntityInfo eInfo) throws RemoteException;
public void removeEntity(String clientId,
  String entityId) throws RemoteException;
```

- **Entity Interaction**. These methods support interaction with the entity's capabilities: getting current context values, entity relationships, content, calling services, subscribing to and retrieving events.

```
// content
public String getContent(String clientId,
  String entityId, String contentAttr) throws RemoteException;

// events
public EntityEvent[] getEvents(String clientId, int timeout)
    throws RemoteException;
public void subscribe(
  String clientId,
  EventSubscription[] eventSubs, String eventListener)
      throws RemoteException;
public void unsubscribe(
  String clientId, int subId) throws RemoteException;
public int addSubscription(String clientId,
  EventSubscription eventSub)
      throws RemoteException;
public void unsubscribeAll(String clientId)
      throws RemoteException;
public EventSubscription[] getSubscriptions(
  String clientId) throws RemoteException;

// relationships
public String[] getRelatedEntities(String clientId,
  String entityId, String relationship) throws RemoteException;
public void addEntityRelationship(
  String clientId, String entityId,
  String otherEntityId, String relationship)
```

```
    throws RemoteException;

// services
public DataObject callService(
  String clientId, String entityId, String serviceType,
  int serviceId, String methodName, DataObject[] parms)
    throws RemoteException;

// context
public ContextValue getContextValue(
  String clientId, String entityId, String attribute)
    throws RemoteException;
public void setContextValue(
  String clientId, String entityId, String attribute,
  ContextValue context)
    throws RemoteException;
```

- **Introspection and Capabilities** Finally, methods in the last category support finding entities by (RDF) type, or using a SPARQL [103] select clause, and discovering the current context attributes, service interfaces and events exposed by an entity.

```
// Introspection and capabilities
public String[] findEntities(
  String clientId, String selectWhere) throws RemoteException;
public String[] findEntitiesByType(
  String clientId, String entityType) throws RemoteException;
public String getEntityDescription(
  String clientId, String entityId) throws RemoteException;
public String[] getContextAttributes(
  String clientId, String entityId) throws RemoteException;
public String[] getImplementedServices(
  String clientId, String entityId, String serviceType)
    throws RemoteException;
public String[] getEventsFired(
  String clientId, String entityId) throws RemoteException;
```

To access objects in other subsystems, the Façade methods access static methods in the UIF object as shown in Figure 4.3. Note that while applications may be permitted to add new entities, or set context, they can't themselves access, add or

remove new *components* into the system. This is the role of *adapters*, described in Section 4.8. Methods in all four categories of the Façade are delegated to objects in the Environment Composition Logic described next.

4.4 Environment Composition Logic

The Environment Composition Logic subsystem contains three key interfaces – the `Environment`, `Entity`, and the `ClientManager` as shown in Figure 4.3. `Environment` and `Entity` objects handle all access to the model, integrated systems and components, while the `ClientManager` manages application authentication and subscriptions to asynchronous events.

4.4.1 Environment and Entity Interaction

A Façade call that interacts with the environment as a whole, such as `findEntities` or `addEntity`, will delegate to an instance of `Environment`. This call will then query the Model and Reasoning subsystem described in Section 4.5.

A Façade call that interacts with a single entity, using methods such as `getContextAttributes` or `callService`, will first *bind* an Entity to a resource id in the model. Binding associates the Entity with a resource in the model encapsulated by a `ModelEntity` object. The Façade then calls methods on the Entity instance, which in turn queries the model through the `ModelEntity`. Wrapping the model in interfaces in this fashion allows our queries and model implementations to change without affecting the ECL.

To interact with a component (either hosted by the UIF or in an underlying system) the `Entity` object will first query the Model for the component that handles the request (i.e., an event subscription, service call, or context query). The `Entity` method then delegates the request to either a Native Component, or the Mes-

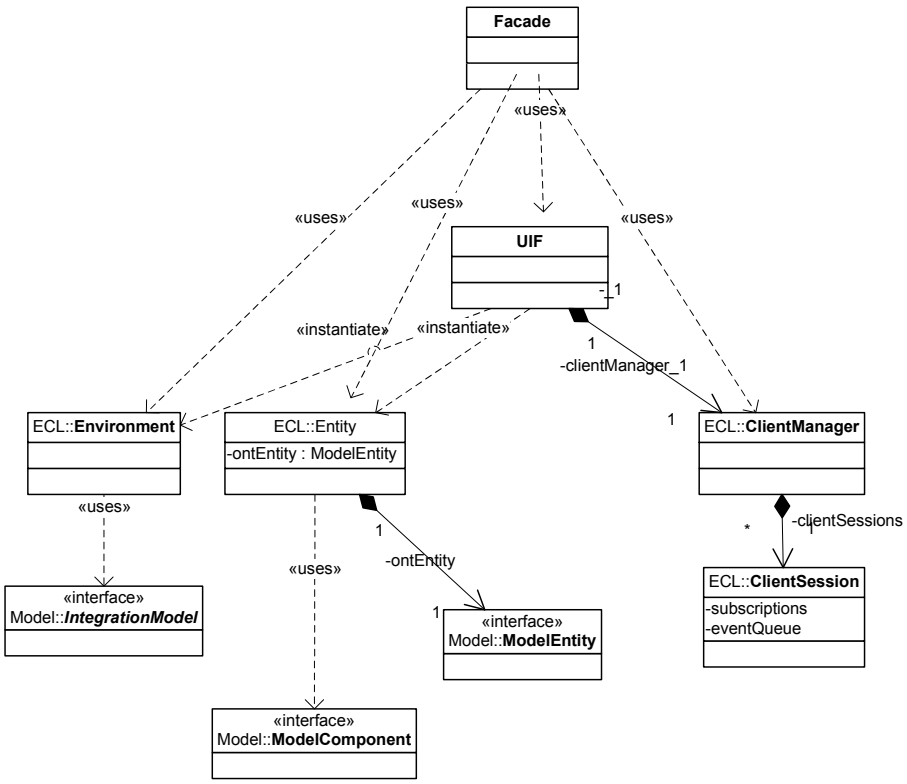


Figure 4.3: Key classes of Environment Composition Logic.

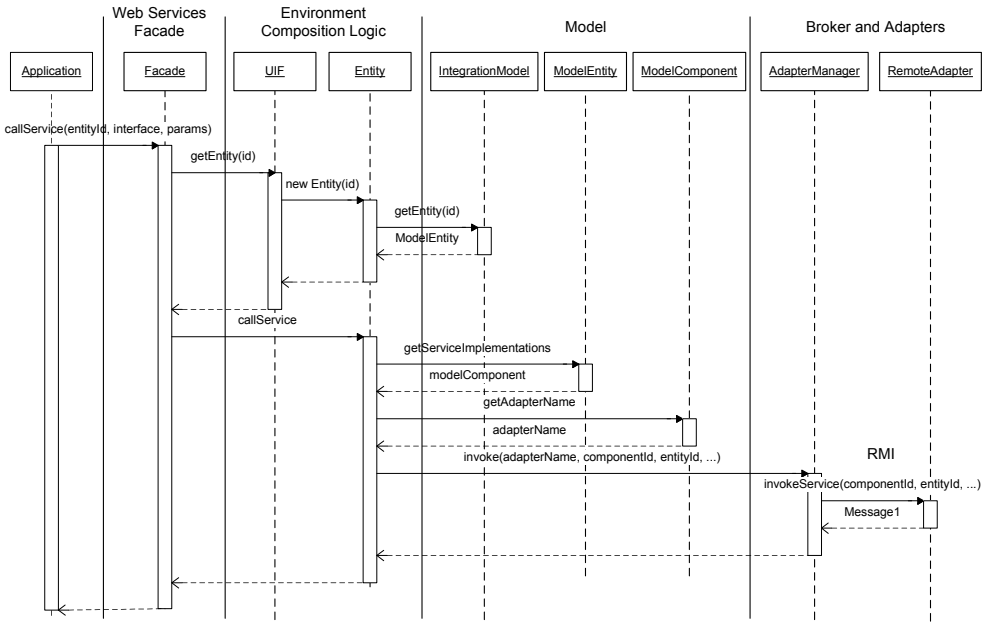


Figure 4.4: Sequence diagram for Facade.callService().

sage Broker. A simplified sequence diagram for the Façade `callService` method is shown in Figure 4.4.

In addition to illustrating the message flow, this diagram highlights potential sources of overhead incurred when introducing the UIF between an application and an integrated system. We incur overhead from the Web Service library when it marshals the SOAP call to the corresponding Java method. Another potential delay is incurred when the Environment Composition Logic queries the Integration Model. Before it is finally handled by a native ubicomp system, it must be delegated by the Broker, and processed by the appropriate remote adapter. These sources of overhead are discussed in more detail in Chapter 5, Section 5.4.3.

4.4.2 Application and Subscription Management

The UIF ClientManager shown in Figure 4.3 handles application authentication and subscriptions to asynchronous events. From our experience, managing subscriptions for an integrated environment is not trivial since a single composite model event may be signalled by multiple event sources in more than one system. Furthermore, applications must be informed when an event source associated with one or more subscriptions is no longer available.

When an application logs in using the Façade, the client id is validated, and a new ClientSession object is created. ClientSessions contain a list of event subscriptions and a queue of events for application retrieval as shown in Figure 4.3.

When an application subscribes to an event, an EventSubscription is added to the ClientSession. An EventSubscription contains a unique subscription id, the application client identifier, entity identifier, the event name and a subscription data structure specific to that event type. One event supported by the MUSEcap system [40] (discussed in Section 5.3.4) called a *nearEvent* is fired when users are near particular places or other users. In this case, the *nearEvent* subscription data contains entity ids, and the distance between the entities that causes the event to fire. The subscription is then delegated to the appropriate Entity object and aggregated *EventSource* component(s) using the same pattern as the service call example in Figure 4.4.

If the subscription is destined for an integrated system, the subscription is forwarded to the appropriate adapter by the Message Broker. Since a distributed adapter may fail, our Message Broker called the AdapterManager maintains an adapter-subscription mapping in case an adapter is shut down or fails. When this occurs, the AdapterManager sends an event to the ClientSession (and asso-

ciated application) indicating that the subscription has expired. If an individual *EventSource* in an integrated system is shut down, it is the responsibility of the adapter to signal to the UIF on its behalf. The UIF will then signal application subscribers that the event capability of that entity has been removed.

Note that, in some cases, multiple *EventSource*s may supply the same named *EventType* capability. This is the case when an entity exposes an event such as *contextUpdate* which is signalled when *any* context value changes. In this case a single event subscription to a *contextUpdate* event must be dispatched to more than one event source, potentially implemented in multiple systems. Because of this, event subscriptions are reference counted in the UIF. They can be removed from a `ClientSession` only when there are no remaining event sources holding the subscription.

When an `Adapter` signals an event on behalf of an integrated system, it supplies the client id of the subscribing application, and an `EntityEvent` data structure that includes the subscription id, the event name, the entity id of the entity that exposes the event and the event-specific data. The client id is used to look up the `ClientSession` object in the ClientManager so that the event can be relayed to the application. An application supplies a subscription id to unsubscribe from an event. The unsubscribe call is dispatched to the `AdapterManager`, removed from its adapter-subscription list, then forwarded to the appropriate `Adapter`.

To summarize, The `ClientManager` component of the Environment Composition Logic authenticates applications and tracks their subscriptions to events. Overall the Environment Composition Logic is responsible for querying the Model and Reasoner to delegate calls, including event subscriptions, to the appropriate integrated system.

4.5 Model and Reasoning

The Model and Reasoner subsystem manages the current environment model including all entities, their capabilities and aggregated components. It typically includes some static context values, in particular those that are not supplied by any adapter or internal component, but may be useful to applications. This includes the static relationships between locations and their coordinates for example. We elected to store entities, components, and static entity relationships and context in the model; dynamic entity relationships and context requests are delegated to an adapter. With many entities and rapidly changing context, we found that it was not practical to retrieve or cache current context values in the integrated model for entity discovery. The core of our model is implemented using the Jena semantic web framework [1]. We integrated the general purpose reasoning engine supplied with Jena into the UIF.

Upon start up, the current model can be loaded into RAM from a standard relational database, or from RDF files supplied by an integrator. The RDF contains the UCM core model, specializations of the model for particular environments or required features, and static instance data such as static locations, users, objects, and fixed components. Once the knowledge base is loaded, rules to support the UCM and custom rules supplied by an integrator are then read into the system. Custom rules may be used to automatically aggregate certain fixed components with certain entities. For example, a server currently supplies user identity and presence information for users in the model; context sources for these capabilities are automatically associated with *CampusUser* entity types using the following rules:

```
# all CampusUsers have presence context
[userPresence: (?entity ucm:aggregates pmedia:presenceSource)<-
    (?entity rdf:type campus:CampusUser)]
```

```
# all CampusUsers have position context
[userPosition: (?entity ucm:aggregates pmedia:positionSource)<-
    (?entity rdf:type campus:CampusUser)]
```

Our initial model implementation wrapped a single Jena inference model: a data model wrapped with a reasoning engine and associated rules. Unfortunately, the reasoner supplied with the framework is not thread safe, so that all queries and changes to the model must be serialized. For small prototype models, we found performance to be adequate, but with larger models we needed another approach. While it should be possible to improve performance by reducing the number of rules and triples in the model to only those required by our application, we wanted to maintain the flexibility of the Jena reasoner and use OWL as much as possible. To address this we created an `IntegrationModel` implementation that maintains two cached query models. One is available for queries at all times, while the other is updated in the background when a change is made. Once all changes have been processed, the background model is then swapped in and used for queries. Using this implementation, both queries and changes are fast, however, changes to the model are not returned in queries until some time after a change is made.

We wrapped all queries to the model using three interfaces: the `IntegrationModel`, `ModelEntity` and `ModelComponent`. Wrapping the integration model in these interfaces allowed us to change not only the integration model as described previously, but also the queries used without changes to other parts of the platform. An `IntegrationModel` is responsible for model initialization and model composition using methods to add and remove entities and components. It contains methods to find entities by type, or for maximum flexibility, query for entities using a SPARQL [103] clause.

Together these interfaces provide the ECL with the knowledge required to dis-

patch method calls to the right adapter and/or associated UCM components. Components by be handled by an Adapter as described in Section 4.8 or by native UIF components described next.

4.6 Component Container

The native Component Container subsystem is a simple micro-container for UCM components implemented by an integrator. Such components can provide composite services, or custom context sources that make use of one or more integrated ubicomp systems, or services only available to the integration platform.

The key classes and interfaces for the Native Component subsystem are shown in Figure 4.5. The `ComponentFactory` provides methods to instantiate components by providing the component identifier and the Java class for the component, both from the UCM. (The Java class is the value of a *binding* UCM *Component* property.)

The `ComponentRegistry` encapsulates a mapping from UCM `componentId` to `Component` objects for lookup by other parts of the system. Native components implement a Component interface, or one of its derived interfaces: `ServiceHandler`, `ContextSource`, `EventSource`, `EntityHandler` and `Environment`. Each of these corresponds to a UCM component type, other than the `Environment`. The `Environment` is a special component that will receive `addEntity` and `removeEntity` method calls from the Façade to handle adding and removing entities to the model that are not handled by a supporting system and so must be added or removed by the UIF itself. As their names imply, `ServiceHandler` implements a method to invoke its services, a `ContextSource` supports context queries by providing an attribute. A `ContextSource` is also an `EventSource` that signals events to objects that implement `EntityEventListener` interface. Note that `ClientSes-`

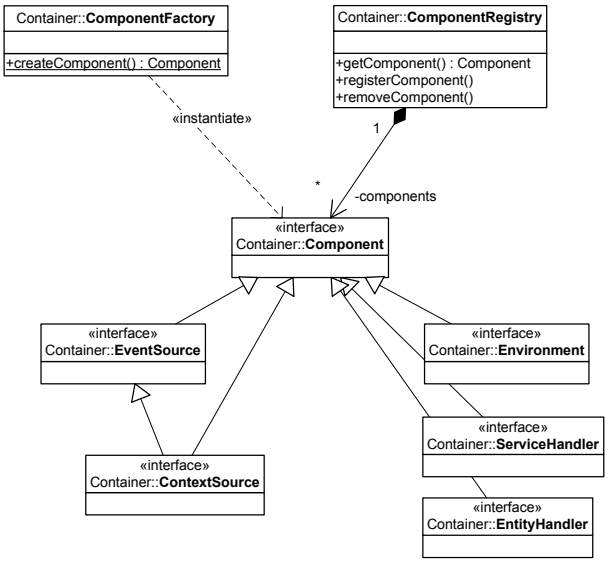


Figure 4.5: Key classes and interfaces of Component Container subsystem.

sion objects in the ECL are also Components managed by the Container. They implement the `EntityEventListener` interface so they can receive events from both Adapter-hosted components and native UIF components in the Container.

One composite service we have implemented provides a broadcast message service associated with a place that calls individual message services for those users *containedIn* a place. Another provides the capability to track users' locations supplied by GPS-equipped PDAs. Other native components may provide custom context inference services, or integration with components exposed using Web Services outside the UIF.

A native component implements one of the above interfaces, typically by specializing one of several abstract classes supplied with the system such as `AbstractContextSource` and `AbstractServiceHandler`. The components register themselves by providing their unique identifiers, their Java binding class and whether

the component should be instantiated on start up or on demand. An example native component declaration follows:

```
<campus:CampusPositionSource rdf:ID="positionSource">
  <ucm:binding>ca.ubc.cs.uif.prototype.PositionContextSource
  </ucm:binding>
  <ucm:onStartup>true</ucm:onStartup>
  <ucm:hasContextType rdf:resource="&ucm;pointLocation"/>
  <ucm:firesEventType rdf:resource="&ucm;contextChangedEvent"/>
</campus:CampusPositionSource>
```

The *positionSource* native component implements a Web Service to receive GPS coordinate updates from PDAs in the field. The UCM id of this native component is *positionSource*. Its Java binding is the `PositionContextSource` class. The *onStartup* property is true, indicating that it should be instantiated when the system starts up. It exposes a *pointLocation* context type, and a *contextChangedEvent* event type.

To summarize, the Container subsystem contains UCM components implemented within the UIF by an integrator. These components can provide composite services or custom context sources for example either making use of an integrated system or providing new functionality not anticipated by an integrated system. Like components supplied by an integrated system, Container-hosted components are also registered with the Model and Reasoner subsystem.

4.7 Message Broker: AdapterManager

The `AdapterManager` object in the UIF is our message broker responsible for managing adapters and forwarding method calls using Java Remote Method Invocation (RMI) [38]. The `AdapterManager` is also responsible for maintaining adapter leases and tracking event subscriptions for adapters in case an Adapter shuts down. Remote Adapters first register with the AdapterManager, using the

`adapterStarted()` method of AdapterListener shown in Program 4.1. Once an adapter is registered with the system, the `AdapterManager` dispatches method calls destined to Components associated with it. Adapters add and remove entities and components from the model depending on the integrated system as described in the next section.

4.8 Adapters

Key to our approach to integration is the use of *adapters* which sit between the UIF and an underlying ubicomp system. The Adapter interface shown in Program 4.2 is designed to encapsulate the functionality of an existing ubicomp system. Adapters ensure the integration framework holds the exposed entities and capabilities of the integrated system, maintain mappings between components, event subscriptions and entity identifiers, and marshal method calls to and from the integrated ubicomp system on demand. Adapters initiate a connection with the UIF by calling the `adapterStarted` method of the AdapterListener interface shown in Program 4.1.

Program 4.1 AdapterListener interface.

```
public interface AdapterListener extends Remote {
  void adapterStarted( String adapterName, Adapter adapter);
  void fireEvent(String adapterName, String sourceId,
  String subscriberId,EntityEvent event);
  String addEntity(EntityInfo eInfo);
  void removeEntity(String entityId);
  String addComponent(ComponentInfo cInfo);
  void removeComponent(String componentId);
}
```

The `add/removeComponent` and `add/removeEntity` methods are called by Adapters to add and remove entities and components to the model as they are discovered in an integrated system. Aggregation links between entities and com-

ponents may be established by the adapter in the ComponentInfo data structure, or specified in an integration rule installed in the framework. Adapters signal events by calling the AdapterListener `fireEvent` method.

Program 4.2 Adapter interface.

```
public interface Adapter extends Remote {
  boolean start(boolean reset);
  void stop();
  boolean check();
  ContextValue getContextValue(
    String componentID, String entityId, String attribute);
  void setContextValue(
    String componentID,String entityId, String attribute,
    ContextValue value);
  DataObject invoke(
    String componentId, String entityId, String serviceType,
    String methodName, DataObject[] inArgs);
  int[] subscribe(
    String componentId, String subscriberId,
    EventSubscription[] eventSubs);
  public void unsubscribe(
    String componentId, String subscriberId,
    EventSubscription[] eventSubs);
  String[] getRelatedEntities(
    String componentId, String id, String relationship);
  void addEntityRelationship(
    String componentId, String id, String entityId,
    String relationship);
  public void addEntity( EntityInfo info);
  public void removeEntity( String entityId);
}
```

Several method calls support direct interaction with the underlying system and correspond directly to those in the interaction calls of the Façade. Note that these method calls include a component id used to identify the particular component: EntityHandler, Service, ContextSource or EventSource, in the underlying system.

Two Adapter method calls, `addEntity` and `removeEntity` are called when an

application adds or removes an entity from the model. This can occur when an application registers a new user with the model, or when a new place of interest is added for example. The new entity information is broadcast to all adapters in case they need to update their native model with this new information.

4.8.1 Adapter Framework

To facilitate adapter development, we created a lightweight adapter framework. The key objects of this framework are shown in Figure 4.6.

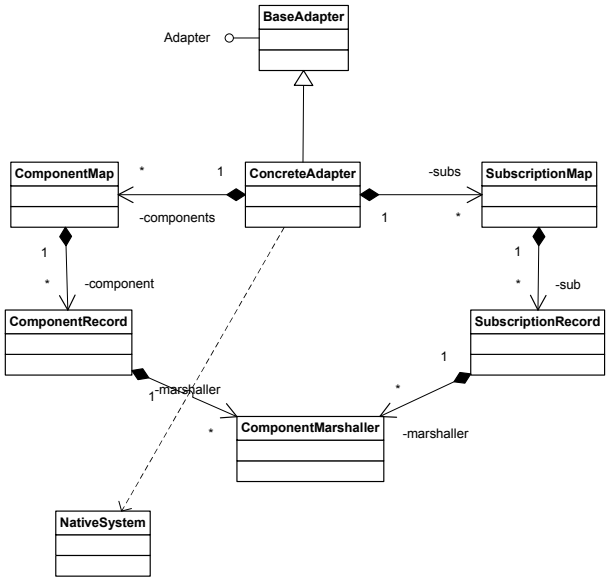


Figure 4.6: Key classes of the Adapter framework.

The adapter framework consists of the `BaseAdapter` abstract class which handles set up of a remote RMI-based adapter, and initial registration with the UIF. A `ComponentMap` contains `ComponentRecords` indexed by native component id and UCM id. A ComponentRecord includes a ComponentMarshaler which marshals data to and from the native system format.

Fixed components such as the iROS *Event Heap* are added to the model when the adapter starts up using the `addComponent` call. Dynamic components such as Context Toolkit *Widgets* are added to the composite model when the adapter receives an event from the native system. Since some systems do not support such events, some adapters will poll for newly discovered components.

When an application makes a synchronous entity interaction call, the Adapter looks up the `ComponentRecord` based on the UCM component id supplied. It then gets the `ComponentMarshaler` with that record, uses it to marshal the data and then makes a native system call. The `ComponentMarshaler` is subclassed to perform the necessary data and protocol marshaling on a per-system, and if necessary, a per-component basis.

To maintain subscriptions for asynchronous events, an adapter typically uses a `SubscriptionMap`. This maintains a mapping of UIF subscriberId and componentId to native event subscriptions. When a new subscription is received, the subscription data is marshaled to the native system subscription, a native `subscribe` call is made and a new `SubscriptionRecord` is added to the map. Like a ComponentRecord, the `SubcriptionRecord` contains a `ComponentMarshaler` for the native event source. When an event is received by the integrated system, the Adapter looks up the corresponding `SubscriptionRecord`. The associated `ComponentMarshaler` is then used to marshal the event data structure to a UIF EntityEvent data structure. The AdapterListener `fireEvent` call is then made to the UIF to forward to the waiting `ClientSession` and associated UIF application. Since the same UIF subscription may correspond to a subscription to multiple components on the same system, the UIF will supply a subscription for each subscriptionId-componentId pair; the `SubscriptionMap` will maintain a record for each subscription-component pair.

Using the adapter interfaces and framework described here, we implemented

a prototype deployment that emulates an campus-scale ubicomp environment by integrating four existing systems [36, 40, 47, 83] described in the next chapter.

4.9 Summary

In this chapter we presented the Ubicomp Integration Framework, a platform for ubicomp systems integration used to evaluate the UCM. We described each subsystem of the UIF including the Façade Web Service, Environment Composition Logic, Model and Reasoning subsystem, Component Container, Broker and Adapter Framework.

To summarize, the UIF manages an environment model repository containing an instance of the UCM. The model repository is queried to dispatch method calls to the appropriate adapter or internal components. In addition, the UIF manages subscriptions for internal or integrated event sources for applications of the integrated environment model. For example, to query for context about an entity (referring to the UIF architecture diagram in Figure 4.1) an application will make a Web Service call to the Façade. This will be delegated to the Environment Composition Logic (ECL) subsystem. The ECL queries the knowledge base hosted by the Model and Reasoning subsystem for the component that supplies the requested context. If the context request can be satisfied by the Model and Reasoning subsystem directly (e.g., static context) the context is returned directly. If not, the ECL examines the component properties to determine whether the query should be handled by an internal UIF component in the Native Component Container, or an integrated system dispatched by the Message Broker. If it is destined for an integrated system, the request is dispatched to the appropriate adapter and marshaled to the system-specific API or protocol. The query response is then returned to the application in the reverse direction. Service calls and event subscriptions follow a

similar call pattern.

In the next chapter we present our experience in creating the composite environment shown in Figure 5.1 to evaluate the UCM and the UIF supporting system.

Chapter 5

Evaluation: An Integrated Campus Environment

To evaluate the Ubicomp Common Model and our approach to ubicomp systems integration we considered three key questions:

1. Can our model be used to support applications that make use of multiple underlying ubicomp environments each with their own different abstractions and programming models?

2. Is our model flexible enough to support all four classes of ubicomp systems identified in Chapter 2: *Component Compositions*, *Context Frameworks*, *Smart Space Systems*, and *Wide Area Systems*.

3. While meeting our first two requirements, can our model be implemented such that it still offers both adequate performance and is practical for application development?

To answer these questions we describe our experience integrating a set of four ubicomp systems into a composite environment for application development. The prototype deployment illustrated in Figure 5.1 was created by integrating the Equip Component Toolkit (ECT) [47] developed at the University of Nottingham, a Component Composition System, the Context Toolkit (CTK) [36] developed at Georgia Tech, a Context Framework, iROS [83] developed at Stanford, a Smart Space

System, and the MUSEcap platform [40] developed at the University of British Columbia, a Wide Area System. MUSEcap is a system designed for campus scale deployments similar to Active Campus [49]. In this deployment the UIF acts as an intermediary between all four ubicomp systems as shown.

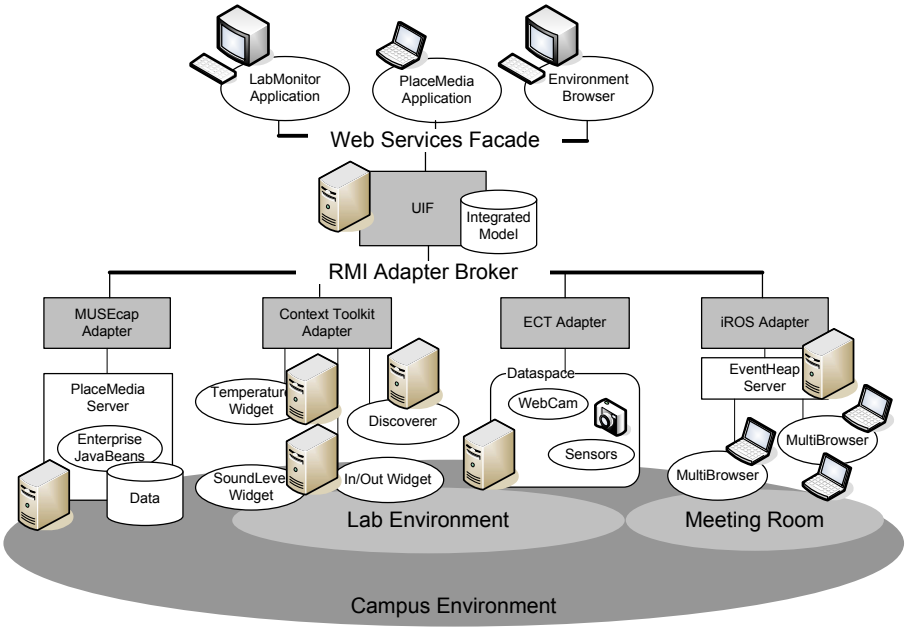


Figure 5.1: Composite campus environment deployment.

5.1 Applications

To exercise our composite environment we developed three prototype applications: PlaceMedia, the Lab Monitor, and the Environment Browser. Rather than inventing our own unique applications, we aimed to support applications inspired by previous work, particularly those used with the systems we integrated.

Each application was developed for a specific purpose. The PlaceMedia ap-

plication described in Section 5.1.1 was used primarily for rapid development and testing of the UCM, UIF and adapter interfaces. The Lab Monitor application in Section 5.1.2 was designed to exercise the capabilities of a composite environment as a whole, making use of four integrated systems simultaneously. Finally, the Environment Browser (Section 5.1.3) was developed primarily as a tool for testing and debugging the integrated environment. Each application is described in more detail here.

5.1.1 PlaceMedia

PlaceMedia was created for rapid development of the UIF and adapter subsystems. To accomplish this, the dependency between the PlaceMedia application and the MUSEcap platform was broken by inserting the UIF system between the application and the MUSEcap platform. This way we ensured that the application APIs were adequate for at least one application, and the Adapter API was sufficient to integrate the MUSEcap system.

PlaceMedia, modelled after the Active Campus Explorer application [50], allows users to see their own location and the location of their friends on a campus map as they roam around. They can communicate with each other, and see the locations of interesting landmarks nearby. PlaceMedia users can see their own location and the location and on line status (presence) of friends using a map-based interface. They can also place *media markers* containing text, images or video clips at places of interest and subscribe to alerts that let them know when they are near another user or media marker. Users communicate with each other using a built-in instant messaging facility.

Location and presence is updated periodically by an agent application running on PDAs or Tablet PCs. Location is derived using wifi signal strength [69] or

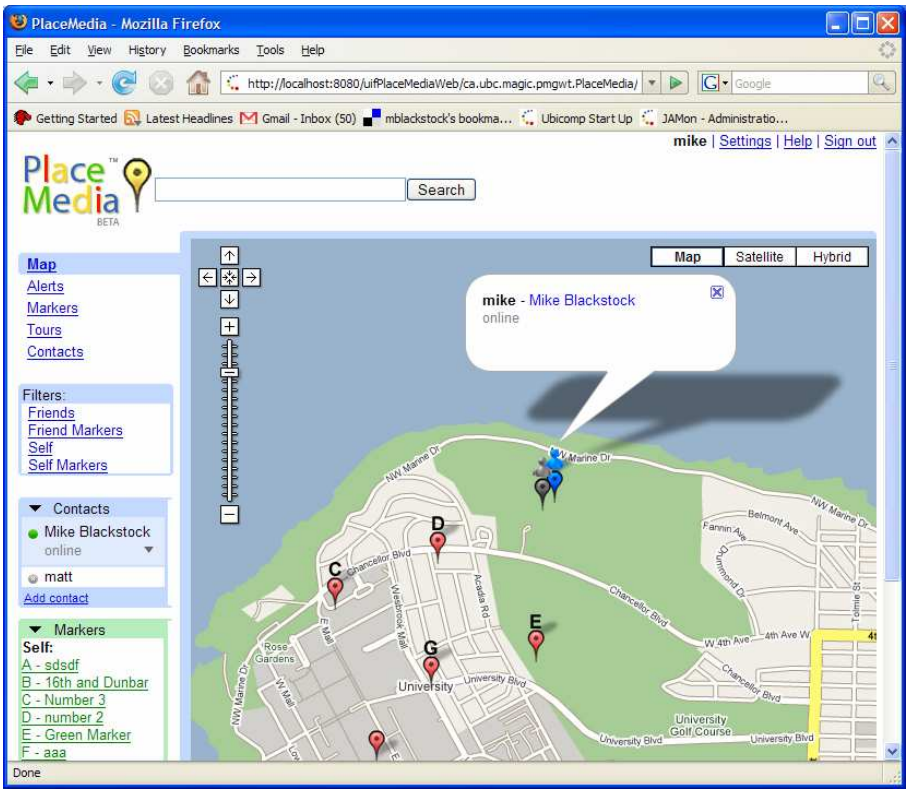


Figure 5.2: PlaceMedia user interface.

provided directly by small Bluetooth GPS units. The web based application user interface was implemented using the Google Web Toolkit[1] and the Google Maps API[2] to display locations and media markers. Since the PlaceMedia application was originally designed to use the MUSEcap platform directly using an Enterprise JavaBeans [75] API, we created an UIF adapter for MUSEcap, and modified the PlaceMedia application to use the UIF Façade web services API. This allowed us to insert the UIF and composite model between the PlaceMedia application and MUSEcap platform for development and testing of both interfaces. The user

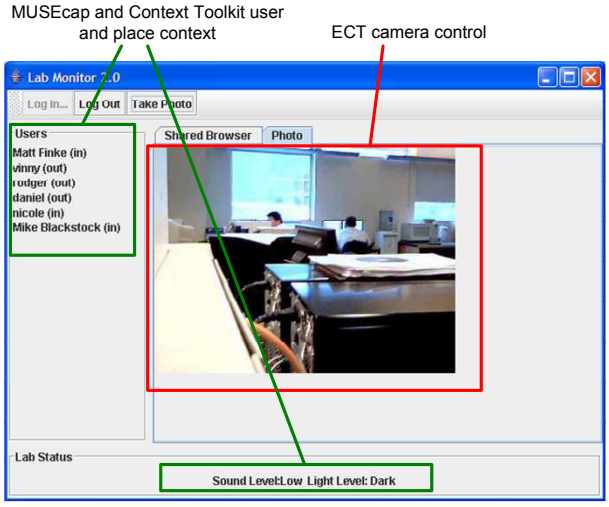
[1] see `http://code.google.com/webtoolkit/`
[2] see `http://code.google.com/apis/maps/`

interface for the tablet PC version of the application is shown in Figure 5.2.

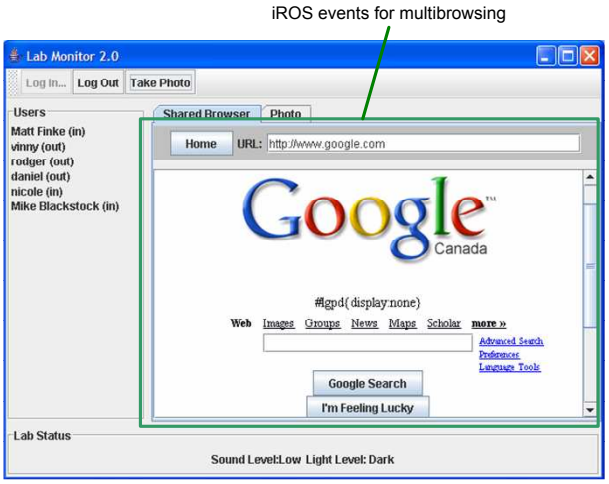
5.1.2 Lab Monitor

The "Lab Monitor" application was designed to exercise the capabilities of a composite environment as a whole, making use of four integrated systems simultaneously. This required integration with three additional existing ubicomp systems as described in Section 5.3.

The features of Lab Monitor were inspired by the Context Toolkit In/Out Box and the iRoom Multibrowse [65] applications. The Lab Monitor user interface, shown in Figure 5.3, performs two primary functions. First, it monitors our lab by providing information about who is present (left pane), the current sound and lighting levels (bottom pane), and allows users to take a photo of the lab as shown in Figure 5.3(a). This allows remote users to see who is present and whether there is a meeting going on, for example. Secondly, it allows users to share web pages with others by broadcasting URLs to other Lab Monitor applications as shown in Figure 5.3(b). The LabMontitor application makes use of features from each of the four underlying systems. User identity is supplied by the UBC MUSEcap system. User presence in the room is supplied by an In/Out widget from the Context Toolkit (CTK) using an RFID sensor. Sound and light levels are also supplied by CTK widgets and appropriate sensors. To share web pages with other users, the UIF broadcasts URLs using the iROS Event Heap. A web camera is controlled by an integrated Equip Component Toolkit component. This integration was accomplished using four adapter implementations described in Section 5.3.



(a) Lab Monitor Photo tab



(b) Lab Monitor Multibrowse tab

Figure 5.3: Lab Monitor application user interface. The indicated systems provide different capabilities to the application.

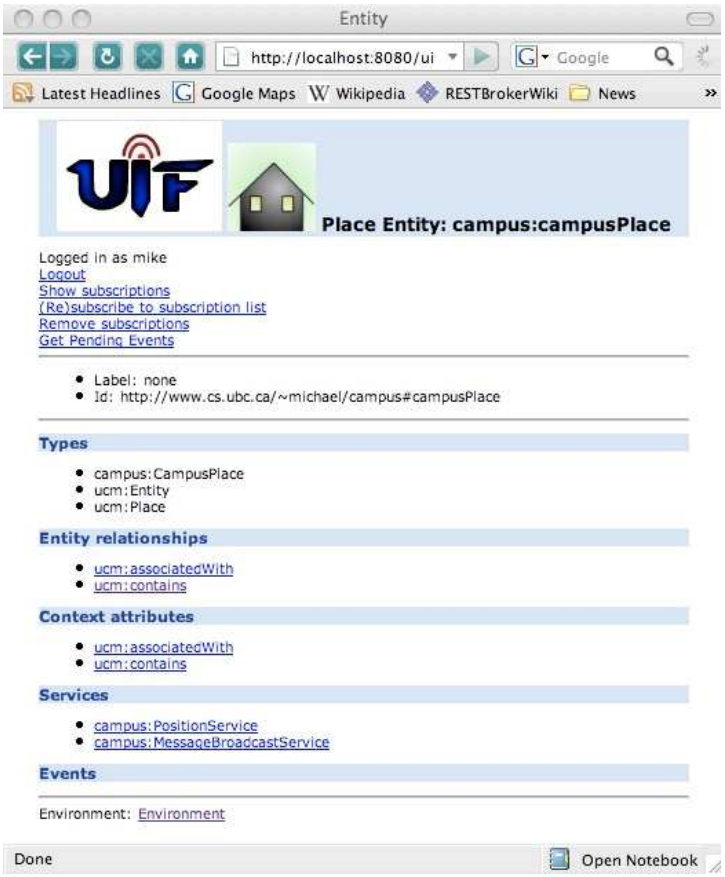


Figure 5.4: Environment Browser user interface.

5.1.3 Environment Browser

To interactively explore the contents and capabilities of the integrated environment model, the Environment Browser is supplied with the UIF system. While this application was designed more as an administrative and development tool, it also serves as a demonstration of how an application can browse and interact with an environment, independent of the types of hosted entities and associated capabilities.

The Environment Browser is a web application where each page corresponds to an entity in the model (person, place or thing). The page contains links corresponding to the current context, entity relationships, and events associated with that entity. Users can navigate to other entities following entity relationship links, retrieve current context values, call services and subscribe to events associated with entities. The user interface of the Environment Explorer is shown in Figure 5.4.

5.2 System Integration

We tested our applications using a composite Campus environment composed of four integrated systems [36, 40, 47, 63]. In this section we describe this composite model and the design and implementation of each adapter.

5.2.1 Campus Composite Environment Model

The composite model in our prototype integration deployment is summarized in Figure 5.5. It consists of a root *Environment* entity which hosts a static *Campus* entity. By doing this, the model anticipates incorporating other places outside the campus. The Campus contains *Users* and *PlaceMarker* entities dynamically supplied by the MUSEcap system. The campus also contains a static *MAGIC Lab* entity, and a *Meeting Room* place entity as shown. Users may be *friends* with each other and may be *contained in* either the Campus or the MAGIC Lab.

Capabilities such as context, services and events are associated with users and place markers. User capabilities are supplied by MUSEcap and the Context Toolkit. The MAGIC Lab contexts sound level and temperature are supplied by the Context Toolkit; the photo service is supplied by an ECT component. Meeting Room capabilities for sending URLs to the browser services and subscribing to and sending interaction events are supplied by the iROS system. Note that these

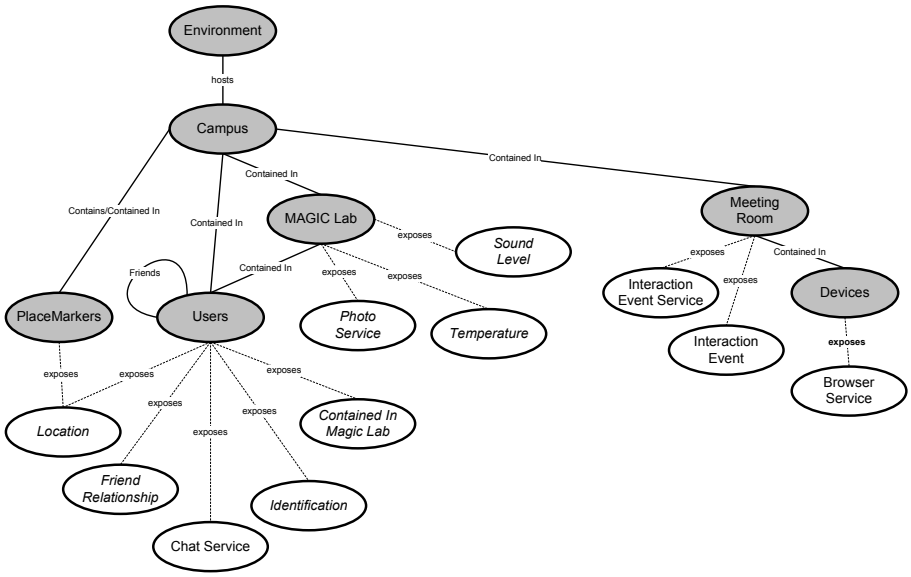


Figure 5.5: Composite environment model including entities and key capabilities.

capabilities will only appear when the underlying adapter, system and components are available.

Each adapter design and implementation is described next.

5.3 Adapter Design and Implementation

For each adapter in this section, we provide a table that maps the UCM core abstractions (environment, entities, context, services, event, content) to facilities in the integrated system. We then highlight key implementation decisions that informed our general adapter design process and integration lessons learned. Following a description of our adapters, we summarize the adapter design process, outlining the steps required by an integrator to integrate a ubicomp system using the UCM with the UIF.

5.3.1 Equip Component Toolkit Adapter

The Equip Component Toolkit is a platform which aims to support the rapid development of ubiquitous computing experiences. ECT *components* (typically implemented using JavaBeans [96]) are used to interact with devices, or may be software only components for processing. Components define named properties used to control their operation or read their current state. For example, the Camera Component has an *triggerImageCapture* property and an *imageLocation* property. When the value of the *trigger* property is changed, the URL for an image captured by the camera can be retrieved using the *imageLocation*.

Components are managed within local component containers which may be coordinated in a distributed *dataspace* installation [46, 61]. Components may be combined by linking shared properties with similar types to one another so that property changes are propagated between the two installations [47]. A key advantage of using the ECT dataspace and property links as a coordination mechanism is that components can be combined very easily without knowledge of a specific toolkit API. Table 5.1 summarizes how the UCM core abstractions map to a typical ECT deployment.

Design and Implementation

To make use of components in an ECT installation, we created an Adapter component that interacts with the dataspace as a whole, monitoring and interacting with any component, property and links in the space. The Adapter is notified when new components are discovered or removed and it interacts with ECT components on behalf of UIF applications.

In our prototype adapter integrated a web cam ECT component; other components can be integrated in a similar manner. When a component of the type Camera

Table 5.1: ECT Adapter UCM Abstractions

Abstraction	Implementation
Environment Model	Dataspace containing components that publish properties, named, typed values. These component properties may be linked to other component properties with the same type.
Entities	Not supported directly.
Entity Relationships	Not supported directly.
Context	Properties of components can be considered or used as context values.
Services	Services are not supported directly, but a component's property values can be set, and then a trigger property is changed.
Events	Property change events may be monitored.
Data/ Content	Not supported directly, implicit in some components properties. For example, the URL of the last photo taken by the web camera component is in a property.

is discovered, a UCM Service description is added to the model. This service includes `takePhoto` and `getLastPhoto` methods. When the `takePhoto` method is called in the UIF service, this changes the value of the *triggerImageCapture* property of the ECT camera component. This causes a photo to be taken. The photo URL is saved in the Camera *imageLocation* property. When the `getLastPhoto` method is called this value is read from the component property and returned to the UIF application.

5.3.2 Context Toolkit Adapter

The Context Toolkit (CTK) is a well known system that introduced the notion of reusable *Context Widgets* that supply applications with a wide variety of context [36]. Widgets encapsulate sources of context such as sensors or context interpreters and related services. A typical CTK deployment can be mapped to core UCM abstractions as summarized in Table 5.2.

Table 5.2: Context Toolkit Adapter UCM Abstractions

Abstraction	Implementation
Environment Model	A Discoverer component tracks CTK components available to applications. The resources available to an application may come and go as components in the environment are started and stopped.
Entities	Entities are often handled explicitly by context aggregators that aggregate context about a certain entity such as a user, place or activity. Attributes of a widget may specify the static locations or user that the context or services potentially apply to. Similarly dynamic context values can represent entities such as users and locations.
Entity Relationships	Entity relationships are not supported as a first class abstraction in the CTK, however, certain widgets can associate entities such as places and users. For example the in/out widget associates users with the place that the widget serves.
Context	Context supported by the toolkit include simple context values for the ambient sound level, lighting and temperature level in a room.
Services	Context widgets can also support associated services. For example a light widget could also support lighting control.
Events	A common event in the toolkit is support for context changes. Applications can subscribe to these events, specifying the context attributes they wish to monitor.
Data/ Content	Content is often treated as context in the context toolkit. For example, Questions, and presentation content in the Conference Assistant are context associated with a Presentation room or activity [37].

Design and Implementation

We focused on integrating a deployment of the CTK by accessing its Discoverer and certain context widgets directly. Since the source code available on Sourceforge[3] did not support our sensor hardware[4], or was unavailable, we re-implemented several widgets described in Context Toolkit publications ourselves [34, 36]. Specifically we implemented a light sensing, ambient sound level, and temperature sensing widget and a 'presence' widget [89] that sends events when an RFID tag is passed over a reader. An in/out widget was also implemented making use of the

[3] `http://contexttoolkit.sourceforge.net/`
[4] `http://www.phidgets.com/`

presence widget and an Interpreter component to track the state of certain users when they enter or leave a room by waving the RFID over the reader. Since CTK components may come and go at any time, the CTK adapter subscribes to the Discoverer component, indicating the components that it can integrate into the composite environment. When a supported component is discovered, its RDF description is injected into the composite model by the adapter, and a mapping from the UCM component id to a CTK identifier is maintained.

For simple widgets such as sound, light and temperature sensors, the component mapping from a Context Widget to a UCM *ContextSource*, and associated data marshaling in the adapter was straightforward. Mapping the capabilities of the in/out widget presents some unique challenges, however, since it supports the containment relationship between a room and a few users equipped with RFID tags. Since the in/out adapter only applies to certain users in our composite model (i.e., only those with RFID tags), we had the CTK adapter supply the entities aggregated by this component explicitly when the component description is added to the system. To make it easy for applications to find users that can be *contained in* a specific room, we marked these users as *pmedia:MeetingUser*s in the model. The RDF *roomLocationSource* component describing the in/out widget is shown in Program 5.3.2 including descriptions of the users of type *pmedia:MeetingUser* that have RFID tags.

Our in/out widget supports queries for the relationship *user contained-in place*, but not the inverse relationship *place contains users*. Because of this, the corresponding UCM component is aggregated by user entities, but *not* place entities.

The in/out widget can signal events when a user enters or leaves the room. The event data supplied when this occurs contains the user identifier and in/out state. Since the event is related to the place not an individual user, this event is best aggregated by the room it serves. As a result we used two UCM component

Program 5.1 Component description for the roomLocationSource Context Source.

```
<ucm:ContextSource rdf:ID='roomLocationSource'>
  <ucm:adapter>contextToolkit</ucm:adapter>
  <ucm:hasContextType rdf:resource='ucm;containedInType' />
  <ucm:aggregator rdf:resource='&pmedia;mike'/>
  <ucm:aggregator rdf:resource='&pmedia;rodger'/>
  <ucm:aggregator rdf:resource='&pmedia;nicole'/>
  . . .
</ucm:ContextSource>

<rdf:Description rdf:about='&pmedia;mike'>
  <rdf:type rdf:resource='&meeting;MeetingUser'/>
</rdf:Description>
<rdf:Description rdf:about='&pmedia;rodger'>
  <rdf:type rdf:resource='&meeting;MeetingUser'/>
</rdf:Description>
<rdf:Description rdf:about='&pmedia;nicole'>
  <rdf:type rdf:resource='&meeting;MeetingUser'/>
</rdf:Description>
```

descriptions for the CTK in/out widget, a context source and event source for a single CTK component. The first exposed the *contained-in* relationship, handling relationship queries for user entities; the second exposed a *relationship-changed* event handling subscriptions to these events for the place. There need not be a one to one correspondence between UCM component descriptions and the components of an integrated system.

CTK widgets support the same event type for almost all widgets: the UPDATE event signalled when context changes. When an entity in the model aggregates context from several CTK widgets, a single UIF subscription for context changes must therefore be propagated to several CTK components. Consequently, there is not necessarily a one to one correspondence between UIF subscriptions and CTK subscriptions.

5.3.3 iROS Adapter

The iROS meta-operating system was constructed to support experiments around interaction with large screen displays in interactive workspaces [63]. Several prototype versions of the installation called the iRoom were set up. From this experience researchers at Stanford identified some of the most important characteristics of an interactive workspace infrastructure [83]. A key subsystem underlying many of the other iROS facilities is the Event Heap [64]. This provides a coordination mechanism that decouples applications and services from one another, allowing the system as a whole to be more stable. On top of the Event Heap, several other general purpose facilities are available. The ICrafter service infrastructure provides service discovery and interaction layered over the Event Heap, and the Data Heap provides a facility for storage of files associated with a place, independent of how they are stored. Arbitrary meta data associated with content in the Data Heap can be used by applications directly or by the Data Heap itself to transform data from one format to another. In Table 5.3 we outline the mapping from the UIF abstractions to facilities in iROS.

Design and Implementation

For this integration task we focused on integrating a representative ICrafter service, and support for the Event Heap coordination facility to ensure the UCM provides adequate coverage of these capabilities. One application and ICrafter service supplied with the iROS installation is the Multibrowse application and *Butler* service. The Butler is used to control applications like Internet Explorer on a PC. Multibrowse allows Internet Explorer users in the workspace to send links to other PCs in the workspace to share URLs. We described the ICrafter Butler service using a UCM *URLService* service interface that implements a `sendURL` method. With

Table 5.3: UCM Abstractions Mapped to the iROS System

Abstraction	Implementation
Environment Model	The ICrafter subsystem models the environment as a set of resources such as application services, components and associated state. The Data Heap contains relevant content and associated meta data. The Event Heap is used as a communications mechanism.
Entities	Entities information, such as devices, users, groups, are found in Event Heap event fields, which may be associated with ICrafter services and state.
Entity Relationships	Not supported directly, but group membership can be inferred from information in Event fields (group, user). ICrafter service descriptions can include location of services.
Context	Variables in the state space. For example, a light switch state
Services	ICrafter services
Events	Events are a first class abstraction in the iROS system. Arbitrary events can be produced or consumed by an iROS client application or subsystem.
Data/Content	The data heap provides storage for content.

this service integrated, UIF applications, like iROS Multibrowse, can open Internet Explorer to display a URL on any iROS equipped PC. Initially it was unclear whether this service should be aggregated by a *user* entity, or a *device* entity in the room. Since the device name is included in the service description, and there is no way of knowing from iROS who the current user of a PC is, we decided to introduce a laptop device entity into the environment when a Butler/URLService is added to the model.

To explore the suitability of the UCM in supporting the core capabilities of the Event Heap, we needed a strategy to both receive and send events. To accomplish this, we described the Event Heap as a single UCM component with two capabilities. The first is a *meeting:InteractionEventService* that allows applications to inject arbitrary events into the Event Heap. The second is the *meeting:InteractionEventType* with the event name *meeting:interactionEvent* to support subscription to arbitrary event heap events. Our current implementation is limited

to subscribing to events by the *type* field only, but could be extended to support templates as in the Event Heap API. These capabilities are exposed by the meeting room when the component corresponding to the iROS adapter starts up. With this capability we were able to support the broadcasting and receiving of URLs to share web pages in our Lab Monitor application through the Event Heap like other iROS applications.

5.3.4 MUSEcap Adapter

The MUSEcap system was developed at UBC to facilitate the development of campus-scale ubiquitous computing applications. Like the PlaceMedia application described previously, MUSEcap was also used for rapid UCM and UIF development, in this case, for the adapter interfaces outlined in Section 4.8. MUSEcap interfaces are exposed as a type of Enterprise JavaBean called a *Session Bean*. Session Beans are a type of server-side component used in Java-based transaction monitors typically used to implement application logic. As in previous system adapters, we first mapped the core UCM abstractions to a typical MUSEcap deployment as summarized in Table 5.4.

Design and Implementation

In integrating MUSEcap, we aimed to leverage its facilities to manage users and their context between indoor campus locations. Another unique facility offered by MUSEcap was its ability to add (register) and remove entities such as people and places called Place Markers to an environment model and fire events when users are near places of interest or other users.

To integrate these capabilities, we created entity types and capability descriptions for the context, event types and service interfaces exposed by MUSEcap APIs.

Table 5.4: UCM Abstractions Mapped to the MUSEcap System

Abstraction	Implementation
Environment Model	The MUSEcap environment model is implemented using a database. This database is wrapped by several service interfaces (Session Beans) to access information about users and markers, their relationships and context.
Entities	Entities supported include users and media markers, or places marked with a latitude and longitude. User entities can be added by the application when new users register, and marker entities can be added by users.
Entity Relationships	Users are related to each other using a 'Roster' in the database indicating that the first user in the table is a friend with the second. Users may also 'own' media markers, indicating that the user created a marker for others to see.
Context	Context supported includes user location updated by agent applications running on PDAs or Tablet PCs. Context also includes user identity, and presence information such as 'on line', 'off line' or 'away'. Marker context includes its location, and identity. In PlaceMedia, content such as text, or an image or video can be associated with a marker. We also consider this marker context.
Services	Users in the system can send messages to each other. We consider this to be a service associated with those users.
Events	The system can alert applications when users are within a certain range of each other, or when a user is in range of a marker. Event subscriptions specify the two users, or the user and marker and the range. Events are signaled only once when they are in range.
Data/ Content	Content such as text, images or video can be associated with a marker. We consider marker content as a form of context.

Since all components of MUSEcap are available as long as the server is running, static component information is provided in the configuration files loaded by the UIF on start up. Several rules were supplied to associate MUSEcap components to Place Marker and User entity types. For example, one rule specifies that if an entity is a *pmedia:CampusUser*, it aggregates the *PositionSource* component. Since the PostionSource exposes a *location* ContextType, all *pmedia:CampusUsers* also expose the ability to retrieve their location.

When the UIF calls the adapter `start` method, the MUSEcap adapter updates

the UIF with the entities (users and markers) and their static context values, creating unique UCM ids for each entity. A mapping from UCM entity identifiers to MUSEcap identifiers is maintained by the adapter using a database. When new users or markers are added, the UIF model is updated, and new mappings are created. To interact with the system, the UIF supplies the UCM entity id and component id. In the adapter, the component id is used to determine the method to call in the MUSEcap API.

Unlike other integrated systems, MUSEcap supports the ability to add or remove new users and places to the composite model. To make use of this feature, UIF applications such as PlaceMedia will call `addEntity` or `removeEntity` using the Façade Web Services interface. Unlike other methods that are targeted to specific systems, these method calls are then relayed to *all* Adapters in case they need to add/remove these entities to/from their native environment model. When the MUSEcap adapter is called, the UIF entity data structure is transformed to MUSEcap data, and the appropriate calls are made to the MUSEcap API.

Finally, since the MUSEcap platform assumes applications periodically poll for events, the MUSEcap adapter, taking on the role of an application on behalf of all UIF applications polls for *near* events once a subscription is received from any UIF application.

5.3.5 Adapter Implementation Summary

Table 5.5 summarizes the adapter functionality we implemented as described previously, categorized by the UCM abstractions.

Clearly we did not attempt to map all of the available functionality of the chosen platforms. Rather, our efforts focused on exercising our UCM abstractions to gain a better understanding of the integration development process, abstraction

Table 5.5: Adapter Implementations by UCM Abstraction

System/ Abstraction	Context Toolkit	Equip Component Toolkit	iROS	MUSEcap
Environment Model	Discoverer	Equip Dataspace	ICrafter sub system, and EventHeap	SessionBean interface to database
Entities	Static locations, users	Implied place (lab) where components are located	Host device in ICrafter service description	Place markers, users
Entity Relationships	InOutWidget relates places to users.	Not supported	Host device contained-in the meeting room	User friends, place marker ownership
Context	User location, user presence, room sound, light level, temperature	Component properties (not implemented)	iROS State API (not implemented)	User location, user identity, presence, place marker location
Services	Context Widget Services (not implemented)s	Camera Service	Browser service, interaction event service	Chat service
Events	Relationship changed, context changed	Property changes (not implemented)	Interaction event	User or place marker Near event
Data/ Content	(Not supported)	Get photo service method	DataHeap (Not implemented)	Place marker content

mappings and trade offs such as adapter complexity vs performance, which we report on in Section 5.4.

5.3.6 Adapter Design Process

From our application development and integration experience we have derived a design process for integrating ubicomp systems using the UCM into a single composite model. This process consists of six steps as follows.

1. **Determine the application-environment interaction points.** Interaction points for integration may be an API, protocol, message format, data store, or other abstraction hosted by distributed components or central servers. While locating these interaction points seems like a straightforward step, in some cases, the integrated system's resources may not be designed for easy external application integration. For example, in ECT or iROS the application is assumed to be a component or a composition of components in the system itself. In this case, it may be possible to access integrated components indirectly using the coordinating system such as the ECT data space or iROS Event Heap.

2. **Decide on the environment capabilities to expose to outside applications.** Considerations in this step of course begin with the capabilities available in the ubicomp system deployment to integrate. Capabilities correspond to the sensors, actuators, software services, and context sources available. To avoid unnecessary integration work, an integrator should consider the functionality required by applications of a composite environment where interoperability is most important. Generally, only a subset of the system's capabilities needs to be exposed to applications using the UCM.

3. **"Find" the *missing* or *implicit* entities in the model and associate capabilities with these entities.** In this step, we make any *implicit* entities, people, places, and things *explicit* in the composite model. In some cases, entities are already explicit in a system. For example, the MUSEcap interfaces expose *user* and *place marker* entities directly. In other cases, entities are only implied. For example, since the camera ECT component itself provides no information about the physical entities it is associated with an integrator may introduce a camera *device* entity to integrate an ECT camera

component. This is necessary since the ECT camera software component itself provides no information about the physical entity (camera or place) it is associated with. Once we've made entities explicit in the model, we must associate the integrated systems' capabilities with these entities. The ECT camera service is associated with the camera device; an instance of the iROS Multibrowse shared browser service is associated with a laptop device. Similarly, the Event Heap and a temperature Context Toolkit Widget can be associated with *place* entities.

4. **Encapsulate interaction points of the integrated system in UCM component abstractions.** In some cases, there may be an obvious mapping between a UCM component and an integrated system interaction point. For example, a CTK widget maps naturally to a UCM context source; similarly, an ICrafter service maps to a UCM service component. In other cases, it may be beneficial to create a UCM component abstraction where none exists. For example, the Event Heap is considered to be a central coordination mechanism. To allow applications to inject arbitrary events into the Event Heap, it is described as a component with *ServiceInterface* and *EventType* capabilities. The service interface provides a method to inject events, the event type allows outside applications to subscribe to interaction events.

5. **Create rules or explicit aggregation relationships.** To associate components and their implemented capabilities to adapters, aggregation relationships can be added to the model directly when components are discovered, or can be inferred by rules that link entities to components based on type or static context stored in the model. An integrator can use either technique.

6. **Implement the Adapter.** The last step is to implement the adapter. In our

prototype work, we attempted to leverage code from a previously developed adapter in new ones. Over time we recognized that common facilities of an adapter emerged that could be used in an adapter framework as discussed in Section 4.8.1.

5.4 Evaluation

In this section, we address the three questions posed at the beginning of this chapter. First, we comment on our experience in using the UIF to build applications that use multiple ubicomp systems at the same time. We then discuss adapter complexity to get a handle on the flexibility of our model in supporting the four systems we integrated, each from a different category of ubicomp system as outlined in Chapter 2. We then measure the performance of the system as a whole, to understand the feasibility of our approach. Finally we make some observations on the use of the UIF as a stand alone system for application development.

5.4.1 Application Development

Our experience in using a single API to interact with multiple systems has several advantages. Developers need only learn and use one set of abstractions, and only one API instead of four or more. This should reduce the learning curve and increase the portability of applications. However, these benefits come at a cost: the performance overhead associated with the use of meta-middleware like the UIF and the development of flexible adapters to maintain the integrated model and marshal method calls to and from the integrated systems. In a typical deployment, we expect that UCM application developers will be largely isolated from the cost of implementing adapters since they can be created independently.

5.4.2 Adapter Complexity

To evaluate how well our UCM abstractions capture those of an underlying system we considered the complexity of adapter development. We found that the process of developing became easier with more experience and as previous implementations were refactored for greater reuse. We estimate that the time required to integrate basic functionality (less than 6 components) of a Java-based system was about 2 weeks. This development time depends on the programming model and API, the documentation available, and the capabilities to be integrated. Our first adapter for MUSEcap was created without any shared code and is about 1200 lines of code. The CTK, iROS and ECT adapters share about 400 lines of code, and added about 850, 1050 and 550 lines of additional code respectively. Overall we found that the UCM abstractions provided adequate coverage of the underlying systems' capabilities: adapters were straightforward to develop. In future work we intend to gather more evidence to support both conclusions with a wider study involving additional systems and integrators as the platform is made available to other research groups.

5.4.3 Performance

Next, we considered the performance and the overhead of the integration framework. Applications such as PlaceMedia and the LabMonitor will poll the UIF for new context values, or to retrieve events using the web services interface. They then call services, send events or set context based on events received or user input. To gain insight on the system's responsiveness to application requests, we measured the average time taken to get context supplied by the Context Toolkit, to call a service supplied by ECT or send an iROS event at different (aggregate) request frequencies. To do this we created simple simulated applications that poll

once per second, then varied the number of these polling applications between 5 and 80. We then measured the average latency of the synchronous web service calls made through the UIF to three integrated systems, the Context Toolkit, ECT and iROS. The results are summarized in Figure 5.6.

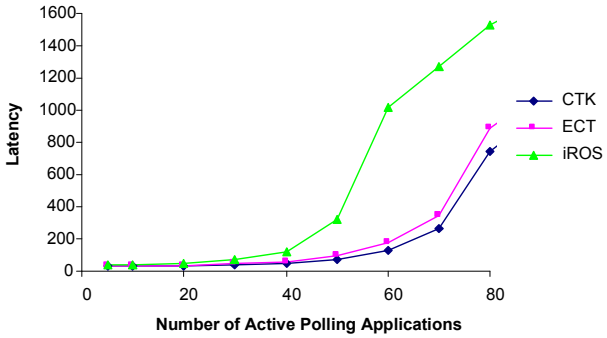


Figure 5.6: Average latency (ms) vs. number of active polling applications (once per second).

In our deployment we hosted iROS, ECT and the CTK on a single 2.13 GHz Pentium Core Duo system with 1 GB RAM, the MUSEcap platform and the UIF in a second 3.4 GHz Pentium D with 2 GB RAM. Simulated applications were run on a tablet PC with 1GB of RAM and a 1.5 GHz Pentium M processor; all machines were on the same LAN. The model consists of 535 data triples; the general purpose rule-based reasoner uses 278 rules. These tests represent a best case response time; before each test we restarted the system and did not change the model. The system was primed with a light test to cache query results. At higher application loads (60-80), we found that the server response time increases to over one second; applications are making requests faster than the server can respond. Overall we found that the system response is less than 100ms for loads of up to 40 or 50 applications polling once per second.

We then examined the overhead of the UIF system in some detail by instru-

Table 5.6: Components of UIF overhead for a call to an ECT component through the UIF framework. These average values are based on 3000 samples taken at about 20 per second.

Component	Average time (ms)	Overall Distribution
Web to Logic Tier	1.096982	3.39%
ECL subsystem	0.491013	1.52%
Model Query	0.955580	2.95%
RMI Broker	2.576806	7.97%
Native ECT call	0.010512	0.03%
Internal time taken	5.130892	15.86%
Web Services[a]	27.214260	84.14%
Total Latency	32.345150	100.00%

[a] calculated by subtracting the measured average internal time from the latency measured by the application

menting key subsystems in the framework and measuring the average time taken for an application to call an ECT service through the UIF. As Table 5.6 indicates, we found that the largest component of overhead was related to the use of the web services middleware and network latency taking more than 84% of the average time taken. Internally our system contributed just over 5 ms to the average time taken to execute a call; most of this time was used by the RMI adapter request and adapter marshaling.

Finally we considered the responsiveness of the UIF while undergoing changes to the model managed by the Jena general purpose rule-based reasoning engine [1]. The composite model changes when applications add new users to the model, or when components are added or removed by an adapter, for example. In this experiment, summarized in Figure 5.7, one application adds then removes a place entity to and from the model every 10 seconds. We measured the latency of a `getContext` made every 2 seconds call from 10 other applications. The latency of the first few calls after a model change increases to more than 1 second then falls

back to under 100ms as shown. After model changes, queries to the model become the largest component of overhead.

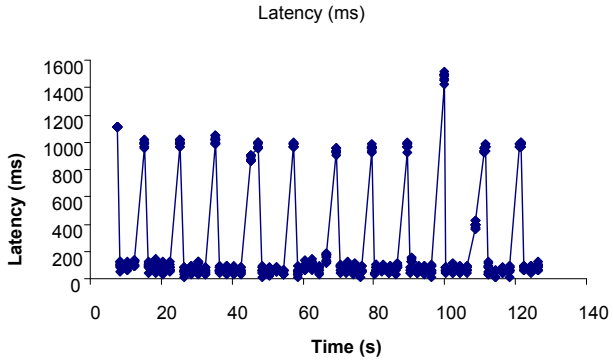


Figure 5.7: Model changes trigger forward reasoning, which causes context requests to wait for more than a second until the reasoner has completed forward reasoning and the write lock is released.

This raised some concerns about the overall scalability of our approach, and in particular the use of an integrated general purpose reasoning engine in our system. To explore how the query and forward reasoning time varies with the model size we conducted two experiments. We first measured the average query time of a *static* model with various model sizes. For this experiment, we added user entities to our model, then measured the average time required to get the context source and associated adapter name associated with a specified entity and context attribute. Our results are summarized in Table 5.7. Overall, we found that the average query time for a *static* model did not change significantly as the model size increased.

Figure 5.7 showed that subsequent queries are delayed until the model is finished updating. This raised some performance concerns. As the size of the model increases, does this delay increase? To answer this question we measured the time taken to complete a request after changes as the model size increases. Our results are summarized in Figure 5.8. The top line indicates the time taken to complete a

Table 5.7: Query Time as (Static) Model Size Increases

Users	Model Size	Time (ms)
10	580	0.516
50	1020	0.500
100	1570	0.516
200	2670	0.547
300	3770	0.750
400	4870	0.578
500	5970	0.781
600	7070	0.609

change to the model, while the bottom line is the time taken to complete a query once an update to the model is complete. Changing the model can take several seconds, and varies linearly with the size of the model, whereas the query time, once a change is complete, is very fast and relatively invariant as the number of triples increases.

Since we expect model changes to occur when new components and entities are added and removed, and in some cases when context is added to the model for reasoning, this will add significant time to application queries that must wait for changes to complete. To address this, we decided to use a "double buffer" scheme to maintain two models in the UIF. One model containing all raw and deduced data is for fast queries (called the query model), while the other is used to collect changes to the model (called the change model). When a change is made to the change model, the system starts reasoning in a background thread by querying the change model for all deduced facts. Once this query is complete, a new query model containing the raw data and deduced facts from the change model is created. At an appropriate time, this new query model containing new data is swapped in replacing the previous query model.

Using this approach both queries and changes to the model are fast, however,

changes to the model will not be reflected in subsequent queries until after a new query model is generated. With large models this can take several seconds as illustrated in Figure 5.8. We found this acceptable for our deployments, but aim to explore other ways to optimize our model implementation in future work.

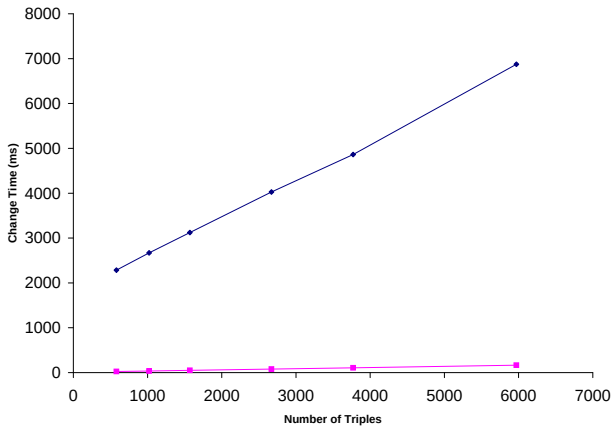


Figure 5.8: Time required to update a model after a change.

Performance Implications

On one hand, we found that the *internal* performance of the system was adequate, but the overall performance factoring in the cost of web services middleware, possible wide area network latency, and the use of an "off the shelf" reasoning engine may seem discouraging. However, with some additional optimizations these issues can be readily addressed. We found that with a suitable model caching scheme and the use of background reasoning it was possible to trade off query and model change times for freshness of the environment model. By choosing light-weight protocols, and by providing APIs to batch more than one context or service call at once, the cost of web services calls can be reduced. Based on our experience with web-based UCM/UIF applications, we have found that a single server-based

application can serve multiple users in a single UIF application, reducing the load on the integrated model.

5.4.4 The UIF as a Stand Alone System

For effective integration of a wide variety of systems ranging from Component Compositions to Smart Spaces, it was necessary to support abstractions of an underlying system, but to compensate for *missing* abstractions. To this end, some of the missing functionality of an underlying system can be implemented directly using the UIF. For example, an integrator can add static entities and relationships found to be useful abstractions but missing from systems such as ECT or only implicitly supported in iROS event fields. "Native" UCM services can broadcast messages to users by calling single users messaging services. Our experience has demonstrated that a system designed around the core abstractions of the UCM may not only serve as an integration platform, but could serve as the basis for a reference ubicomp system implementation. We aim to explore this possibility in collaboration with other practitioners in the ubicomp systems community.

5.5 Lessons Learned

Based on the integration experience described in this chapter, we discuss our findings with respect to the use of the core Ubicomp Common Model that we believe will be useful for both ubicomp systems developers and integrators.

A comprehensive and flexible environment model aids application resource discovery. It is important for an environment model to not only contain components (as in the ECT Dataspace, or the CTK Discoverer) but also entities, relationships, context, and entity capabilities. These abstractions in a model provide a closer

mapping between the "real world" and the digital world coordinated by the ubicomp system making it easier to find relevant computing resources.

Maintaining consistency between a composite model and integrated systems' is challenging. Keeping a composite environment model in the integration system that to some extent mirrors all or parts of the environment model of integrated systems allowed us to unify the the computing resources of more than one system in a consistent and unified manner. This benefits the application developer since they need not concern themselves with how capabilities are provisioned. Moreover since certain abstractions in the UCM such as entities and relationships may be missing or only implicitly supported in an underlying system, this approach allows us to compensate for missing abstractions in an integrating system.

That said, we found that it can be challenging to maintain consistency between an underlying system's model and the composite model. In our deployment we decided to store all entities, components, static relationships and context in the model, and delegate dynamic relationship and context requests to the underlying system. This limits the ability for applications to discover entities by context and relationships. Furthermore, without this information the UIF is unable to infer higher level context and new relationships. While an integrator could configure the UIF to subscribe to context changes or poll for context values periodically, this would force the reasoning subsystem to work continuously to infer new relationships and context values unnecessarily, raising a scalability and performance concern.

To address this we considered delegating discovery requests to the native system. However, this would not only require a common query language to interpret these requests, but also a mechanism to translate these queries to environment discovery requests in the native system. Similarly, results from native discovery requests would need to be collected and interpreted by the integration system, which would require knowledge of the integrated model to interpret them correctly. These

challenges remain an open issue for future research.

An entity abstraction is a natural way to aggregate capabilities: context, service interfaces, events, but there is not a one to one correspondence between entities and components. Based on our experience and others (e.g., [36, 68, 90]) we have found that it is natural to aggregate capabilities such as service interfaces, context and events types around entities, however we found that there is not a one-to-one relationship between entities and *components*, the implementations of these capabilities. More than one entity will often aggregate a component, inheriting its capabilities. For example, a single InOut CTK context widget is associated with both the place entity and the user entities that are tracked. These n-to-n relationships can present challenges for managing event subscriptions where a single UCM event may be supplied by multiple components.

Make entity relationships explicit as they are an important subclass of context. The Cooltown system highlighted the use of entity relationships to create web links between related entities allowing users to browse their changing environment [68]. We found that it was valuable to make these relationships explicit in our integrated model. Entity relationships implicit in iROS event fields or Context Toolkit context values made it easy to find devices in a meeting room or the users in the lab, for example.

Entity type *is an important subclass of context.* The use of ontologies allowed us to easily classify all entities in the system not only as people, places, and things, but also the *types* of people (lab users, students), places (media markers, buildings), and things (laptops, printers). We found that it was natural to aggregate components with certain types of entities. For example, we could aggregate an outdoor location context source with all students, but only aggregate the indoor presence widget with lab users.

Applications are not only consumers of resources, they also manipulate the

model directly. While most systems manage context derived from sensors in the environment, we have found that it is common for applications to not only consume such context, but produce it. For effective integration, application-supplied context must be propagated to the underlying system for use by native applications. Applications are also a source of events; the Lab Monitor application, like Multibrowse [65], can send user interaction events to other UIF applications and iROS services. We found that some applications will also add and remove entities such as places of interest, or newly registered users to an integrated model.

Most ubicomp systems share a common set of events. While several systems have highlighted the importance of event abstractions [83, 87], we have also found that several systems share certain event types. Context Tookit *update* events, like ECT component property changes, are signaled when context values change. When new components are added or removed from a system, the environment model (e.g., Discovery subsystem, Dataspace) will signal applications in case they rely on their capabilities. So far we have found the following high level event types to be common between systems: *context/relationship changed*, *entity added/removed*, *capability changed*. Based on our experience, the consistent implementation of these canonical events will reduce the need for applications to poll for changes allowing applications to more readily respond to changes in the environment.

In general we found that the core UCM abstractions are a superset of the systems' we integrated. This is not surprising since we attempted to develop unified abstractions based on these systems and others. While we found few abstraction mismatches, we did find it necessary to compensate for *missing* abstractions such as entities and entity relationships either in the adapter implementation or the UIF system configuration.

From this experience we came to realize that there are a range of approaches to environment integration: One is to compensate for missing abstractions in under-

lying system(s) as we did. Another is to provide information about the integrated system's capabilities without any compensation for missing abstractions. In the former case, the integration system can provide information about implied entities, interpret context data to create and maintain missing entity relationships, and support composite services. In the latter case, the integration system can simply provide a mechanism for accessing the existing capabilities of an underlying system. An integrator may elect to compensate for some missing abstractions such as static entities but not others. In a sense, our Adapter framework provides a mechanism for exposing system capabilities without (much) compensation for missing abstractions.

Since our aim is to support the full range of integration approaches since we integrate several systems with varying capabilities into a single composite environment, a flexible model for ubicomp environments such as the UCM is required.

5.6 Summary

In this chapter we presented our evaluation of the Ubicomp Common Model. We focused our evaluation efforts on three areas:

- **Application Development.** This experience helped us understand whether our model can be used to support applications that make use of (potentially) several ubicomp environments with varying abstraction levels. We found that the model adequately supported application development, and allowed us to use a single API with more than one system. The experience also highlighted the cost of integration, in both building adapters, and the design and configuration of a composite environment model when using multiple systems.

- **Adapter Complexity.** To gain an in depth understanding of the flexibility of our model, we created adapters for four representative systems, each from a different class of ubicomp system as outlined in Chapter 2. Overall we found that it was possible to create adapters that provided adequate coverage for each systems's capabilities and that these adapters were straightforward to develop.

- **Performance.** To assess the feasibility of the UIF as an integration platform that uses the UCM for environment integration, we measured the performance of the system under both steady state and dynamic environment conditions. In the steady state, we found performance adequate. We then presented an approach to address performance in changing models using a "double buffering" scheme to trade off model freshness for fast query times.

Based on this evaluation experience we discussed some of the lessons learned related to the use of certain UCM abstractions and implementation challenges that can inform the design of future integration systems and ubicomp platforms. In the next chapter we conclude this thesis and discuss future work.

Chapter 6

Conclusions and Future Work

In this thesis we have presented the analysis and design of a core model for ubiquitous computing systems called the Ubicomp Common Model. This model was designed to allow application developers to bridge across a variety of existing ubicomp platforms. To do this, it must adequately describe ubiquitous computing environments in a manner that lends itself to both application portability, specialization to different environment domains, and adaptation to a variety of systems. The UCM design was based on the comprehensive survey of ubicomp systems presented in Chapter 2. From this survey we found that the variety of ubicomp systems abstractions are influenced by the scale of ubicomp deployment and the tradeoffs between making coordinated environments easy to program (high level abstractions) and broad applicability (low level service/component abstractions). We also identified the common abstractions used across systems: an *environment model*, *entities*, *context*, *entity relationships*, *services*, *events* and *data* or *content*.

In chapter 3 we outlined several requirements for a common model for ubicomp and the design of the UCM based on the common abstractions identified in our survey. A key challenge addressed by this model is in finding the right balance between interoperability and suitability for cross domain access while maintaining much of the flexibility of a given underlying ubicomp system. Based on the abstractions identified in Chapter 2 described the UCM design in some detail presenting the three related aspects of the model: the Environment State, Meta State

and Implementation.

The Environment State consists of entities modeled by the supporting system, the relationships between entities and their current context values. The Environment Meta-State aspect is required to support introspection and associates entities with their *capabilities*: the types and quality of events, services, context and content they support. Finally, the Implementation aspect captures *component* abstractions aggregated by entities that provide the services, events, context, relationships and content in an integrated system. We then outline how the UCM addresses the outlined requirements, provided an example of how it can be extended to support security domains and access control, and provided use cases for a *executable* model of the UCM used in a supporting system like the Ubicomp Integration Framework (UIF).

In chapter four we described the UIF: a *meta* middleware system used to evaluate the feasibility of the UCM for both application development and systems integration. We describe the design and implementation of each subsystem in some detail including a description of the use of a integrated knowledge base and reasoning subsystem and *adapters*. The key to our integration approach is to encapsulate integrated system using an Adapter interface. Adapters provide the UIF with information about the resources in the integrated system and handle interaction with the system initiated by UCM applications, converting UIF protocols and data structures to and from those of the native system.

Finally in Chapter 5 we outlined our application and integration experience deploying a composite ubicomp environment that integrated four representative ubicomp systems. From this experience we described our adapter design process and comment on the system's suitability for Application development, integration, performance and suitability as a stand alone system for ubicomp development. We then outlined some lessons learned related to the use of the UCM in this deploy-

ment.

6.1 Lessons Learned

Based on the experience and analysis reported in the previous chapters, we discuss our key findings related to the use of the Ubicomp Common Model for ubicomp systems integration.

6.1.1 A Common Model for Ubiquitous Computing is Useful and Practical

There will always be a variety of ubicomp systems that support various levels of abstraction, and scale. Certain systems will be specialized for different application domains, and continue to track research advances in systems, software engineering and middleware. Despite continuous evolution and the wide variety of systems and application domains for ubicomp systems, we have shown that it is practical to derive a common model for both application portability and systems integration. We have also shown how to instantiate this model in a *meta-middleware* platform for systems integration by composing environments. While there are clear trade-offs in our approach to interoperability in terms of costs of integration in building adapters and configuring a composite environment, performance, and access to the underlying capabilities of the integrated systems, our investigation has shown that these costs are manageable. Clearly our approach of integrating diverse systems using the Ubicomp Common Model is feasible, especially considering the benefits of application portability and interoperability.

6.1.2 Unifying Environment Model is the Key to Integration

Our work has shown that it is feasible for the Ubicomp Common Model and a supporting system to describe and support an environment model that unifies those of a few representative underlying integrated systems. This can be accomplished by replicating key resources and relationships such as entities components and aggregation relationships as in the UIF. We believe that this approach, i.e. providing a unified model in the integration system, is the key to effective integration for two reasons. First, because it exposes a unified and consistent view of the environment facilitating application development and portability. Second, it allows an integrator to compensate for missing abstractions in an integrated system which is critical when attempting to integrate systems that expose lower level abstractions such as service and component compositions with others that expose explicit environment models.

6.1.3 Entity Types and Relationships are Important Subclasses of Context

Based on our analysis of other systems and experience we identified two important subclasses of context that have been shown to be valuable for resource discovery and integration. *Entity relationships* allow an application to more easily find entities and their associated resources. The use of entity *types* also makes resource discovery easier, and enables integration rules to identify entities for the establishment of entity-aggregation-component relationships.

6.1.4 Systems Share Several Common Event Types

We found that several systems share certain types of common events. These included context changed, entity relationship changed, entity added/removed, and

capability changed. Based on our experience, the consistent use of these canonical events increases application portability and reduces the need for applications to poll for changes in the environment.

6.1.5 Applications are Both Consumers and Producers

From our experience we have gained some insight into the dual role of an application in an integrated ubicomp environment. An application is not only a consumer of computing resources (e.g. finding entities, receiving events, retrieving context values, calling services), but also a producer. In the producer role, the application itself is a source of events, content, context and service implementations for other applications in the environment. While we supported this role to some extent through interfaces to set context values, relationships, add and remove entities, in future implementations a general purpose web services interface to the Implementation aspect of the UCM may be provided so that UCM applications can also register their resources with an integrated environment.

6.1.6 Summary

To summarize, we have shown that our approach and model is suitable for integrating a range of representative systems. Although not exhaustive, the systems chosen for composite model integration and application development represent each category of system as presented in Chapter 2: Component Compositions, Context Frameworks, Smart Spaces, and Wide Area Systems. Of course while this does not mean we can support *all* systems, it does indicate that our approach is suitable for a wide range of ubicomp systems to date.

Through the careful analysis of a wide range of ubicomp systems, we have identified the abstractions shared by these systems, and made use of this taxon-

omy in the design of the Ubicomp Common Model. Our integration and deployment experience has informed the design of the UCM, highlighting the advantages and challenges of maintaining a composite environment model, important forms of context and events, and the dual role of an application.

6.2 Future Work

In future work we aim to advance the design of the UCM through continued analysis and practical integration experience. We see opportunities for enhancing and formalizing of the Ubicomp Common Model and continued exploration of the ubicomp systems integration design and implementation space. From our explorations described in this dissertation we believe that a dual approach based on analysis and practical experience is the best way forward.

6.2.1 Enhancing and Specializing the UCM

Based on our integration experience we see opportunities to formalize certain aspects of the core model, and in particular the common event, context and entity types used across systems. We would also like to further explore the development of *Environment Profiles*: specializations of the UCM for certain application and environment domains, perhaps consisting of groups of overlapping ontologies for context, entity types, services suitable for the home, office, and other places.

6.2.2 Security

As discussed in Section 3.5, addressing security challenges will be a challenge that needs to be addressed for cross domain ubicomp deployments (e.g. [41, 93]). In an integration platform like the UIF, we must ensure access to computing resources are protected either individually or in groups that may not correspond to physical

entities, geographical or network boundaries. Future integration platforms must support a variety of access control mechanisms, from traditional name, password and shared keys to more lightweight mechanisms that support more spontaneous interactions involving the application directly.

6.2.3 Improved Scalability

We see opportunities to address scalability concerns in a composite (integrated) environment through optimization of the reasoning subsystem. Approaches include the use of faster general purpose reasoners, special purpose reasoners tuned to our knowledge base, and more efficient model representations in the integration systems. Furthermore, we believe that it may be possible to federate or cluster integration systems to share the work of providing access to a composite model across several servers.

6.2.4 Improved Application Interface

With the development of Web 2.0 applications and the use of dynamic web pages and Asynchronous Javascript and XML (AJAX) techniques we see the increasing use of domain specific HTTP and XML protocols rather than SOAP-based web services. Based on recent experience with other platforms [39] we believe that such protocol for integrated ubicomp environments may be a better fit than Web Services standards in some situations.

Our experience also highlighted the cost of cross domain interaction. To make the most of each web service call we believe it is important to provide interfaces that lend themselves to batch processing. This may include interfaces to get all of the context associated with an entity, or subscribe to all user entities in a given place for example.

6.2.5 Applications as Components

Finally, we believe that the environment interface should be extended or complimented with an interface that supports the dual role of an application as both a producer and consumer of computing resources. While the Facade interface does well in supporting the consumer role, additional interfaces are required to support applications as producers of context, services, content and events.

6.3 In Conclusion

In this work we leveraged the considerable experience in developing ubicomp systems to date to find some common ground for application portability and systems interoperability in ubicomp through the design of the Ubicomp Common Model. The feasibility of using the core UCM was tested using the Ubicomp Integration Framework to integrate representative systems under a composite environment model. Our hope is that systems designers will continue to "climb on the shoulders of giants" in their quest for ubicomp systems interoperability and portability required for applications to be truly ubiquitous.

Bibliography

[1] Jena, a semantic web framework for Java. http://jena.sourceforge.net/. last checked: 9-June-2008.

[2] Universal Description, Discovery and Integration Version 2 OASIS Standard. http://http://www.oasis-open.org/specs/index.php#uddiv3.0.2. last checked: 9-June-2008.

[3] OZONE - new technologies and services for emerging nomadic societies. http://www.hitech-projects.com/euprojects/ozone/, 2004. last checked: 9-June-2008.

[4] SWeDE: Semantic Web Development Environment. http://owl-eclipse.projects.semwebcentral.org/, 2005. last checked: 11-June-2008.

[5] JBoss web services. http://www.jboss.org/jbossws/, 2007. last checked: 9-June-2008.

[6] Eclipse.org Home Page. http://www.eclipse.org/, 2008. last checked: 11-June-2008.

[7] JBoss home page. http://www.jboss.com/, 2008. last checked: 9-June-2008.

[8] MySQL Home Page. http://www.mysql.com/, 2008. last checked: 9-June-2008.

[9] The Protégé ontology editor and knowledge acquisition system. http://protege.stanford.edu, 2008. last checked: 11-June-2008.

[10] WebServices - Axis. http://ws.apache.org/axis/, 2008. last checked: 9-June-2008.

[11] M. Addlesee, R. Curwen, S. Hodges, J. Newman, P. Steggles, A. Ward, and A. Hopper. Implementing a sentient computing system. *IEEE Computer*, 34(8):50–56, 2001.

[12] Gustavo Alonso. *Web services : concepts, architectures and applications*. Data-centric systems and applications. Springer, Berlin ; New York, 2004.

[13] Apple, Inc. Networking - Bonjour. `http://developer.apple.com/networking/bonjour/`, 2007. last checked: 9-June-2008.

[14] Mark Assad, David Carmichael, Judy Kay, and Bob Kummerfeld. PersonisAD: Distributed, active, scrutable model framework for context-aware services. In *Pervasive Computing (Pervasive 2007)*, pages 55–72, Toronto, Canada, 2007.

[15] Jakob E. Bardram. The Java Context Awareness Framework (JCAF) - a service infrastructure and programming framework for context-aware applications. In *Pervasive Computing and Communications (PerCom 2005)*, pages 98–115, Munich, Germany, 2005. Springer.

[16] Jakob E. Bardram, Thomas R. Hansen, Martin Mogensen, and Mads Soegaard. Experiences from real-world deployment of context-aware technologies in a hospital environment. In *Ubiquitous Computing (UbiComp 2006)*, pages 369–386, Orange County, CA, 2006.

[17] C. Becker, G. Schiele, H. Gubbles, and K. Rothermel. BASE - a micro-broker-based middleware for pervasive computing. In *Pervasive Computing and Communications (PerCom 2003)*, pages 443–451, Fort Worth, USA, 2003.

[18] Christian Becker, Marcus Handte, Gregor Schiele, and Kurt Rothermel. PCOM - a component system for pervasive computing. In *Pervasive Computing and Communications (PerCom 2004)*, pages 67–76, Washington, DC, USA, 2004.

[19] Gregory Biegel and Vinny Cahill. A framework for developing mobile, context-aware applications. In *Pervasive Computing and Communications (PerCom 2004)*, pages 361–365, 2004.

[20] Michael Blackstock, Rodger Lea, and Charles Krasic. Toward a shared model for wide area interoperability of ubiquitous computing environments. In *System Support for Ubiquitous Computing (Ubisys)*

Workshop at Ubiquitous Computing (UbiComp 2006), Newport Beach, CA, 2006.

[21] Michael Blackstock, Rodger Lea, and Charles Krasic. Toward wide area interaction with ubiquitous computing environments. In *European Conference on Smart Sensing and Context (EuroSSC 2006)*, Enschede, The Netherlands, 2006.

[22] Michael Blackstock, Rodger Lea, and Charles Krasic. Adapting ubicomp systems to a common model. In *Common Models and Patterns for Pervasive Computing Workshop (CMPPC) at Pervasive 2007*, Toronto, Canada, 2007.

[23] Michael Blackstock, Rodger Lea, and Charles Krasic. Managing an integrated ubicomp environment using ontologies and reasoning. In *Context Management and Reasoning (CoMoRea) Workshop at Pervasive Computing and Communications (PerCom 2007)*, New York, 2007.

[24] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF schema. In *The Semantic Web ISWC 2002*, pages 54–68. Springer Berlin / Heidelberg, 2002.

[25] Barry Brumitt, Brian Meyers, John Krumm, Amanda Kern, and Steven A. Shafer. Easyliving: Technologies for intelligent environments. In *Proceedings of the 2nd international symposium on Handheld and Ubiquitous Computing*, Bristol, UK, 2000. Springer-Verlag. 743885 12-29.

[26] H. Chen, F. Perich, T. Finin, and A. Joshi. SOUPA: Standard ontology for ubiquitous and pervasive applications. In *International Conference on Mobile and Ubiquitous Systems: Networking and Services*, Boston, MA, 2004.

[27] Harry Chen, Tim Finin, and Anupam Joshi. An ontology for context-aware pervasive computing environments. *Knowledge Engineering Review*, 18(3):197–207, 2003.

[28] Harry Chen, Tim Finin, Anupam Joshi, Lalana Kagal, Filip Perich, and Dipanjan Chakraborty. Intelligent agents meet the semantic web in smart spaces. *IEEE Internet Computing*, 8(6):69–79, 2004.

[29] Stuart Cheshire. Zero configuration networking (Zeroconf). `http://www.zeroconf.org/`, 2007. last checked: 9-June-2008.

[30] Eleni Christopoulou and Achilles Kameas. GAS Ontology: An ontology for collaboration among ubiquitous computing devices. *International Journal of Human-Computer Studies*, 62(5):664–685, 2005.

[31] Roger L. Costello. Web services best practice, summary 3. `http://lists.xml.org/archives/xml-dev/200201/msg00477.html`, January 2002. last checked: 9-June-2008.

[32] Cristiano Andr da Costa, Adenauer Corra Yamin, and Cludio Fernando Resin Geyer. Toward a general software infrastructure for ubiquitous computing. *IEEE Pervasive Computing*, 7(1):64–73, 2008.

[33] Nigel Davies and Hans-Werner Gellersen. Beyond prototypes: Challenges in deploying ubiquitous systems. *IEEE Pervasive Computing*, 1(1):26–35, 2002.

[34] Anind K. Dey. *Providing Architectural Support for Building Context-Aware Applications*. Phd thesis, Georgia Institute of Technology, 2000.

[35] Anind K. Dey and Gregory D. Abowd. Toward a better understanding of context and context-awareness. In *CHI 2000 Workshop on the What, Who, Where, When, and How of Context-Awareness*, The Hague, The Netherlands, April 2000.

[36] Anind K. Dey, Daniel Salber, and Gregory D. Abowd. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applictions. *Human-Computer Interaction (HCI) Journal*, 16(2-4):97–166, 2001.

[37] Anind K. Dey, Daniel Salber, Gregory D. Abowd, and Masayasu Futakawa. The Conference Assistant: Combining context-awareness with wearable computing. In *Proceedings of the 3rd International Symposium on Wearable Computers*, pages 114–128, Dublin, Ireland, 1999.

[38] T.B. Downing. *Java RMI: Remote Method Invocation*. IDG Books Worldwide, Inc. Foster City, CA, USA, 1998.

[39] Aiman Erbad, Michael Blackstock, Adrian Friday, Rodger Lea, and Jalal Al-Muhtadi. MAGIC Broker: A middleware toolkit for interactive public displays. In *Middleware Support for Pervasive Computing (PerWare) Workshop at Pervasive Computing and Communications (PerCom 2008)*, pages 509–514, Hong Kong, March 2008.

[40] Matthias Finke, Michael Blackstock, and Rodger Lea. Deployment experience toward core abstractions for context aware applications. In *2nd European Conference on Smart Sensing and Context (EuroSSC)*, Kendal, UK, 2007. Springer.

[41] Bryan Ford, Jacob Strauss, Chris Lesniewski-Laas, Sean Rhea, Frans Kaashoek, and Robert Morris. Persistent personal names for globally connected mobile devices. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, Washington, November 2006.

[42] A. Fox, A. Fox, B. Johanson, P. Hanrahan, and T. Winograd. Integrating information appliances into an interactive workspace. *IEEE Computer Graphics and Applications*, 20(3):54–65, 2000.

[43] A. Friday, N. Davies, N. Wallbank, E. Catterall, and S. Pink. Supporting service discovery, querying and interaction in ubiquitous computing environments. *Wireless Networks*, 10(6):631–641, 2004.

[44] Erich Gamma. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley professional computing series. Addison-Wesley, Reading, Mass., 1995. Erich Gamma ... [et al.]. ill. ; 25 cm.

[45] Paul Grace, Gordon S. Blair, and Sam Samuel. A reflective framework for discovery and interaction in heterogeneous mobile environments. *SIGMOBILE Mob. Comput. Commun. Rev.*, 9(1):2–14, 2005.

[46] Chris Greenhalgh. EQUIP: a software platform for distributed interactive systems. Technical Report 02-002, Mixed Reality Laboratory, University of Nottingham, 2002.

[47] Chris Greenhalgh, Shahram Izadi, James Mathrick, Jan Humble, and Ian Taylor. ECT: a toolkit to support rapid construction of ubicomp environments. In *System Support for Ubiquitous Computing (Ubisys) Workshop at Ubiquitous Computing (UbiComp 2004)*, Nottingham, UK, 2004. Springer.

[48] R. Grimm, J. Davis, E. Lemar, A. MacBeth, S. Swanson, T. Anderson, B. Bershad, G. Boriello, S. Gribble, and D. Wetherall. System support for pervasive applications. *ACM Transactions on Computer Systems*, 22(4):421–486, 2004.

[49] William G. Griswold, Robert Boyer, Steven W. Brown, and Tan Minh Truong. A component architecture for an extensible, highly integrated context-aware computing infrastructure. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 363–372, Washington, DC, USA, 2003. IEEE.

[50] William G. Griswold, Patricia Shanahan, Steven W. Brown, Robert Boyer, Matt Ratto, R. Benjamin Shapiro, and Tan Minh Truong. ActiveCampus: Experiments in community-oriented ubiquitous computing. *Computer*, 37:73–81, 2004.

[51] Tao Gu, Hung Keng Pung, and Da Qing Zhang. Toward an OSGi-based infrastructure for context-aware applications. *IEEE Pervasive Computing*, 3(4):66–74, October-December 2004.

[52] V. Haarslev and R. Möller. Racer: A Core Inference Engine for the Semantic Web. *Proceedings of the 2nd International Workshop on Evaluation of Ontology-based Tools*, pages 27–36, 2003.

[53] S. Harris and N. Gibbins. 3store: Efficient Bulk RDF Storage. *Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems (PSSS03)*, pages 1–20, 2003.

[54] Andy Harter, Andy Hopper, Pete Steggles, Andy Ward, and Paul Webster. The anatomy of a context-aware application. In *MobiCom '99*, pages 59–68, New York, NY, USA, 1999. ACM.

[55] Karen Henricksen and Jadwiga Indulska. A software engineering framework for context-aware pervasive computing. In *Pervasive Computing and Communications (PerCom 2004)*, page 77, Los Alamitos, CA, USA, 2004. IEEE.

[56] Karen Henricksen, Jadwiga Indulska, and Andry Rakotonirainy. Modeling context information in pervasive computing systems. In *Pervasive Computing (Pervasive 2002)*, Zurich, Switzerland, 2002. Springer.

[57] Fritz Hohl, Uwe Kubach, Alexander Leonhardi, Kurt Rothermel, and Markus Schwehm. Next century challenges: Nexus: an open global infrastructure for spatial-aware applications. In *MobiCom '99*, pages 249–255, New York, NY, USA, 1999. ACM.

[58] J. I. Hong and J. A. Landay. An architecture for privacy-sensitive ubiquitous computing. In *Moble Systems, Applications and Services (MobiSys 2004)*, Boston, MA, USA, 2004.

[59] Jason I. Hong. Context Fabric: Infrastructure support for context aware systems. In *CHI '02 extended abstracts on Human factors in computing systems*, Minneapolis, Minnesota, USA, 2001. ACM.

[60] A. Huang, B. Ling, J. Barton, and A. Fox. Making computers disappear: Appliance Data Services. In *MobiCom '01*, Rome, Italy, 2001.

[61] Jan Humble, Andy Crabtree, Terry Hemmings, Karl-Petter Akesson, Boriana Koleva, Tom Rodden, and Par Hansson. Playing with the bits - user-configuration of ubiquitous domestic environments. In *UbiComp 2003*, Seattle, WA, USA, 2003.

[62] Valrie Issarny, Daniele Sacchetti, Ferda Tartanoglu, Francoise Sailhan, Rafik Chibout, Nicole Levy, and Angel Talamona. Developing ambient intelligence systems: A solution based on web services. *Automated Software Engineering*, 12(1):101–137, 2005.

[63] B. Johanson, B. Johanson, A. Fox, and T. Winograd. The interactive workspaces project: experiences with ubiquitous computing rooms. *IEEE Pervasive Computing*, 1(2):67–74, 2002.

[64] Brad Johanson and Armando Fox. The Event Heap: A coordination infrastructure for interactive workspaces. In *Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*. IEEE Computer Society, 2002. 837560 83.

[65] Brad Johanson, Shankar Ponnekanti, Caesar Sengupta, and Armando Fox. Multibrowsing: Moving web content across multiple displays. In *Ubiquitous Computing (Ubicomp 2001)*, pages 346–353, London, UK, 2001. Springer-Verlag.

[66] Dimitris N. Kalofonos, Zoe Antoniou, Franklin D. Reynolds, Max Van-Kleek, Jacob Strauss, and Paul Wisner. MyNet: A platform for secure P2P personal and social networking services. *PerCom 2008*, 0:135–146, 2008.

[67] Apu Kapadia, Tristan Henderson, Jeffrey J. Fielding, and David Kotz. Virtual Walls: Protecting digital privacy in pervasive environments. In *Pervasive Computing (Pervasive 2007)*, pages 162–179, Toronto, Canada, May 2007 2007. Springer.

[68] T. Kindberg, J. Barton, J. Morgan, G. Becker, D. Caswell, P. Debaty, G. Gopal, M. Frid, V. Krishnan, H. Morris, et al. People, Places, Things:

Web Presence for the Real World. In *Mobile Computing Systems and Applications Workshop*, Monterey, CA, December 2000. IEEE Computer Society.

[69] A. LaMarca, Y. Chawathe, S. Consolvo, J. Hightower, I. Smith, J. Scott, T. Sohn, J. Howard, J. Hughes, F. Potter, et al. Place Lab: Device Positioning Using Radio Beacons in the Wild. In *Pervasive Computing (Pervasive (2005)*, volume 3468, pages 116–133. Springer, 2005.

[70] Rodger Lea and Michael Blackstock. Ubisys 2006 workshop report. `http://www.magic.ubc.ca/ubisys/overview.htm`, 2006. last checked: 9-June-2008.

[71] David S. Linthicum. *Next Generation Application Integration*. Addison-Wesley Information Technology Series. Addison-Wesley, 2003.

[72] S. Maffioletti and B. Hirsbrunner. Ubidev: An homogeneous environment for ubiquitous interactive devices. In *Pervasive Computing (Pervasive 2002)*, pages 28–38, August 2002.

[73] B. McBride. Jena: a semantic web toolkit. *Internet Computing, IEEE*, 6(6):55–59, Nov/Dec 2002.

[74] Microsoft. Understanding Universal Plug and Play: A white paper. `http://www.upnp.org/download/UPNP_UnderstandingUPNP.doc`, 2000. last checked: 9-June-2008.

[75] Richard Monson-Haefel. *Enterprise JavaBeans*. O'Reilly, 3rd edition, 2001.

[76] Mark W. Newman, Jana Z. Sedivy, Christine M. Neuwirth, W. Keith Edwards, Jason I. Hong, Shahram Izadi, Karen Marcelo, and Trevor F Smith. Challenge: Recombinant computing and the Speakeasy approach. In *Mobicom '02*, 2002.

[77] Daniela Nicklas, Matthias Grobmann, Thomas Schwarz, and Steffen Volz. A model based, open architecture for mobile, spatially aware applications. In *SSTD 2001: Proceedings of the 7th International Symposium on Spatial and Temporal Databases*, Redondo Beach, CA, USA, 2001. Springer.

[78] Daniel Oberle, Andreas Eberhart, Steffen Staab, Raphael Volz, and In Hans-Arno Jacobsen. Developing and managing software components in an ontology-based application server. In *Middleware 2004,*

ACM/IFIP/USENIX 5th International Middleware Conference, volume 3231 of *LNCS*, pages 459–478, Toronto, Ontario, Canada, 2004. Springer.

[79] Object Management Group (OMG). *The Common Object Request Broker: Architecture and Specification, Version 2.6.1*, May 2002.

[80] Open Services Gateway Initiative Alliance. OSGi Home Page. `http://www.osgi.org/`, 2008. last checked: 9-June-2008.

[81] Organization for the Advancement of Structured Information Standards. OASIS Web Services Security (WSS) TC. `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss`, 2008. last checked: 27-Sept-2008.

[82] Shankar Ponnekanti, Brian Lee, Armando Fox, Pat Hanrahan, and Terry Winograd. ICrafter: A service framework for ubiquitous computing environments. In *UbiComp '01: Proceedings of the 3rd international conference on Ubiquitous Computing*, pages 56–75, London, UK, 2001. Springer.

[83] S.R. Ponnekanti, S.R. Ponnekanti, B. Johanson, E. Kiciman, and A. Fox. Portability, extensibility and robustness in iROS. In B. Johanson, editor, *Pervasive Computing and Communications (PerCom 2003)*, pages 11–19, 2003.

[84] A. Ranganathan, R.E. McGrath, R.H. Campbell, and M.D. Mickunas. Ontologies in a pervasive computing environment. In *Proceedings of the IJCAI-03 Workshop on Ontologies and Distributed Systems*, volume 71, Acapulco, Mexico, 2003.

[85] Anand Ranganathan and Roy H. Campbell. A middleware for context-aware agents in ubiquitous computing environments. In *Middleware 2003*, volume 2672/2003, pages 143–161. Springer Berlin / Heidelberg, 2003.

[86] Hans Gellersen Rene Mayrhofer. Shake well before use: Authentication based on accelerometer data. In *Pervasive Computing (Pervasive 2007)*, pages 144–161, Toronto, Canada, 2007. Springer.

[87] Manuel Roman, Christopher Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell, and Klara Nahrstedt. Gaia: a middleware platform for active spaces. *SIGMOBILE Mob. Comput. Commun. Rev.*, 6(4):65–67, 2002.

[88] W.A. Ruh, W.J. Brown, and F.X. Maginnis. *Enterprise Application Integration: A Tech Brief*. John Wiley & Sons, Inc. New York, NY, USA, 2001.

[89] Daniel Salber, Anind K. Dey, and Gregory D. Abowd. The Context Toolkit: Aiding the development of context-enabled applications. In *CHI*, pages 434–441, 1999.

[90] Bill N. Schilit, Norman Adams, Rich Gold, Michael Tso, and Roy Want. The PARCTAB mobile computing system. In *Fourth Workshop on Workstation Operating Systems (WWOS-IV)*, pages 34–39, Napa, CA, USA, 1993. IEEE.

[91] Bill N. Schilit, Marvin M. Theimer, and Brent B. Welch. Customizing mobile applications. In *USENIX Symposium on Mobile and Location-Independent Computing*, 1993.

[92] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL DL reasoner. *Journal of Web Semantics*, 5(2):51–53, 2007.

[93] DK Smetters, D. Balfanz, G. Durfee, T.F. Smith, and K. Lee. Instant Matchmaking: Simple and Secure Integrated Ubiquitous Computing Environments. In *Ubiquitous Computing (Ubicomp 2006)*, volume 4206, page 477, Orange County, CA, Sept. 17-21, 2006 2006. Springer.

[94] J. Pedro Sousa and David Garlan. Aura: an architectural framework for user mobility in ubiquitous computing environments. In *Proceedings of the 3rd IEEE/IFIP Conference on Software Architecture*. Kluwer, B.V., 2002.

[95] Oliver Storz, Adrian Friday, and Nigel Davies. Towards "ubiquitous" ubiquitous computing: an alliance with the grid. In *System Support for Ubiquitous Computing (Ubisys) Workshop at Ubiquitous Computing (UbiComp 2003)*, Seattle, 2003.

[96] Sun Microsystems. The JavaBeans specification. `http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html`, 1997. last checked: 9-June-2008.

[97] Sun Microsystems. Java 2 Platform, Enterprise Edition (J2EE) Overview. `http://java.sun.com/j2ee/overview.html`, 2008. last checked: 9-June-2008.

[98] Sun Microsystems. Java Enterprise Edition at a Glance. `http://java.sun.com/javaee/`, 2008. last checked: 9-June-2008.

[99] Joo Geok Tan, Daqing Zhang, Xiaohang Wang, and Heng Seng Cheng. Enhancing semantic spaces with event-driven context interpretation. In *Pervasive Computing (Pervasive 2005)*, volume 3468/2005, pages 80–97, Munich, Germany, 2005. Springer. 3468.

[100] Phil Tetlow, Jeff Z. Pan, Daniel Oberle, Evan Wallace, Michael Uschold, and Elisa Kendall. Ontology driven architectures and potential uses of the semantic web in systems and software engineering. `http://www.w3.org/2001/sw/BestPractices/SE/ODA/`, 2006. last checked: 9-June-2008.

[101] W3C. Resource Description Framework. `http://www.w3.org/RDF/`, 2004. last checked: 9-June-2008.

[102] W3C. Web Ontology Language (OWL) overview. `http://www.w3.org/TR/owl-features/`, 2004. last checked: 9-June-2008.

[103] W3C. SPARQL Query Language for RDF. `http://www.w3.org/TR/rdf-sparql-query/`, 2005. last checked: 9-June-2008.

[104] W3C. SOAP version 1.2 part 0: Primer (second edition). `http://www.w3.org/TR/soap12-part0/`, 2007. last checked: 9-June-2008.

[105] W3C. Web Services Description Language (WSDL) version 2.0 part 0: Primer. `http://www.w3.org/TR/wsdl20-primer/`, June 2007. last checked: 9-June-2008.

[106] J. Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, 47(7):76–82, 1999.

[107] Roy Want, Andy Hopper, Veronica Falcao, and Jon Gibbons. The Active Badge location system. *ACM Transactions on Information Systems*, 10(1):91–102, 1992.

[108] Mark Weiser. The computer for the 21st century. *Scientific American*, 265(9):94–104, 1991.

[109] Lawrence Wilkes. ROI - the costs and benefits of web services and service oriented architecture. `http://roadmap.cbdiforum.com/reports/roi/`, 2008. last checked: 9-June-2008.

[110] D. Wood, P. Gearon, and T. Adams. Kowari: A Platform for Semantic Web Storage and Analysis. *Proceedings of the 14th International WWW Conference*, 2005.