AN ARCHITECTURE FOR CONTEXT-AWARE APPLICATIONS

WANG XIAOHANG
(B. Sc., Huazhong University of Science and Technology)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2004

ACKNOWLEDGEMENTS

I wish to extend my sincerest gratitude to my supervisors, Daqing Zhang and Jin Song Dong, for their encouragement, ideas, and support in bringing this work to completion. I am grateful to all my friends in Institute for Infocomm Research, for their help and ideas throughout the course of my work. I would also like to acknowledge the support from the National University of Singapore and Institute for Infocomm Research through a graduate research scholarship that has made my graduate studies and research possible. I would like to thank my family, and my girlfriend, Yuhui, for their understanding and support during the past two years.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ADDITIONAL PUBLICATIONS

1. Xiaohang Wang, Daqing Zhang, Jinsong Dong, Chungyao Chin and Sanka Ravipriya Hettiarachchi. Semantic Space: A Semantic Web Infrastructure for Smart Spaces. In *IEEE Pervasive Computing*, Vol. 3, No. 2, 2004.

2. Xiaohang Wang, Daqing Zhang, Tao Gu and Hung Keng Pung. Ontology-Based Context Modeling and Reasoning Using OWL. In *Workshop on Context Modeling and Reasoning at IEEE International Conference on Pervasive Computing and Communication (PerCom'04)*, 2004, Florida.

3. Daqing Zhang, Zhengning Dai, Xiaohang Wang, Xiao Ni and Song Zheng. A New Service Delivery and Provisioning Architecture for Home Appliances. In *International Conference on Smart Homes and Health Telematics (ICOST2004)*, 2004, Singapore.

4. Daqing Zhang, Xiaohang Wang, Kai Hackbarth. OSGi-Based Service Infrastructure for Context-Aware Automotive Telematics. In *IEEE Vehicular Technology Conference (VTC Spring 2004)*, 2004, Italy.

5. Tao Gu, Hung Keng Pung, Daqing Zhang, Xiaohang Wang. A Middleware for Context-Aware Mobile Services. In *IEEE Vehicular Technology Conference (VTC Spring 2004)*, 2004, Italy.

6. Tao Gu, Xiaohang Wang, Hung Keng Pung, Daqing Zhang. An OWL-Based Context Model in Intelligent Environments. In *Communication Networks and Distributed Systems Modeling and Simulation Conference (CNDS'04)*, 2004, California.

7. Daqing Zhang, Xiaohang Wang, Karianto Leman and Weimin Huang. OSGi-Based Service Infrastructure for Context-Aware Connected Homes. In *International Conference on Smart Homes and Health Telematics (ICOST2003)*, 2003, France.

SUMMARY

The recent convergence of ubiquitous computing and context-aware computing has seen a considerable rise in interest in applications that exploit aspects of the contextual environment to offer services, present information, tailor application behavior or trigger adaptation. However, as a result of the lack of generic mechanisms for supporting context-awareness, context-aware applications remain very difficult to build and developers must deal with a wide range of issues related to representing, sensing, aggregating, storing, querying and reasoning of context. In order to remedy this situation, there is a need for better understanding of the design process associated with context-aware applications, architectural support for the entire context processing flow, and improved programming abstractions that ease the prototyping of applications.

This research in context-aware computing has focused on the architectural support for context-aware application development. This dissertation presents a set of requirements for context-aware applications, based on which we introduced and implemented our architectural support, including an ontology-based context model, a context-aware architecture (namely Semantic Space) and a set of programming abstractions. This architectural support along with an identified design process makes it easier to build sensor wrappers that provide context and context-aware applications that use context. The case study, *SituAwarePhone*, validates our work, and illustrates, in concrete form, the process and issues involved in the design of context-aware software.

CHAPTER 1
INTRODUCTION

1.1 Ubiquitous Computing

The availability and advance of new computing and communication devices, as well as the increased connectivity between these devices are enabling new opportunities for people to utilize computers anywhere and anytime. This recently started era is referred to as Ubiquitous Computing, which was first introduced by Weiser in "The Computer for the 21st Century" [1]. By Weiser, ubiquitous computing aims to enhance computer use by making computers available throughout the physical world, but making them effectively invisible to the user. Ubiquitous computing can be characterized by two main attributes:

- **Ubiquity**: Human-computer interactions are not channeled through a single machine. Access to computation is shift from dedicated computing machinery to ubiquitous computing capabilities embedded in our everyday environments.

- **Transparency**: Computation is non-intrusive and is as invisible and as integrated into the background of the physical environments. To avoid increasingly sophisticated interaction between users and machines, and to allow users to concentrate on their tasks, computation should be distraction-free to the largest extend.

Ubiquitous computing is a phenomenon of incorporating processing and communication capabilities into physical environments. Its major focus is to empower physical environments and to enhance human's interactive experience with them.

1.2 Context-Aware Computing

Ubiquitous computing envisions invisible computing and smooth human-computer interactions. Making computing invisible is not a matter of the physical deployment of computers; it is about how users perceive their interactions with computers. To achieve invisibility in the sense of user's perception, the interaction has to be seamlessly integrated with users' primary task, that is, users can focus on the tasks themselves rather than the interactions with the tools that help them in the tasks.

Typical computer systems are generally conceptualized as input/output systems, which heavily rely on explicit user input. This traditional model is being seen as too restrictive in ubiquitous computing environments: users might have to spend excessive efforts in giving input to multiple devices with heterogeneous interfaces. In contrast, ubiquitous computing systems need to help users achieve their tasks without drawing focus away. When computing and communication power is embedded in the physical world, users, computers, environments, social activities and their relationships with each other can all be rich source of *context* information. Using this information we can make computer systems *context-aware* to minimize or even eliminate explicit user interaction. Therefore, context-aware computing emerged from the field of ubiquitous computing as a technique for imbuing applications with an awareness of their surroundings and situations, in order to achieve transparent interaction.

1.2.1 What is Context?

The term "context" is widely used with different meaning. The following definitions from WordNet [2] provide a generic understanding of the meaning of context in English:

> ***Context:*** 1. *discourse that surrounds a language unit and helps to determine its interpretation*, 2. *the set of facts or circumstances that surround a situation or event.*

In the scope of ubiquitous computing, the understanding of context is different to that in English. Despite of the appearance of context-aware computing in recent years, the concept of context remains ill-defined. Most of the initial efforts for defining context in ubiquitous computing were specific for certain kinds of context, for example, location and time. Schilit et al. [3] claimed that the important aspects of context were the user location and identities of nearby people. Brown et al. [4] and Ryan et al. [5] gave their definition in terms of examples of context information instead of generalizing the concept. Since the number of examples that can be given is limited, the application of this definition is also limited. Context has also been characterized as an applications' environment and situation [6] and as a combination of features of the execution environment including computing, user and physical information [7]. Generalized from previous work [8], Dey offers the following definition, which is perhaps now the mot widely-accepted:

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

In this thesis, we adopt Dey's definition as a reference. Apart from the philosophical discussion on the semantic of context, there are some primitive forms of context that developers agree on. Taking an object-oriented approach, we identify three classes of physical objects (user, location, computing entity) and one class of conceptual objects (activity) that characterize a ubiquitous computing environment. Linked together, these objects form the skeleton of a contextual environment. They also provide primary indices

into associated contextual information, For example, given a location, we can acquire related context such as indoor temperature, noise level, people and activity inside. The identification of primary context forms the basis for our context model (CHAPTER 3).

1.2.2 What is Context-Awareness?

Context-aware computing was first introduced by Schilit and Theimer [3] in 1994 to be software that "*adapts according to its location of use, the collection of nearby people and objects, as well as changes to those objects over time*". Since then there have been numerous attempts to give context-aware computing a definition, most of which were specific to application's characteristics, such as the adaptation, reactivity, responsiveness and sensitiveness to context. For instance, Pascoe et al. [9] define context-aware computing to be the ability of computing devices to detect and sense (sensitiveness), interpret and respond to (reactivity) aspects of a user's location environment and computing devices. In [4], context-aware applications are defined as applications that dynamically change or adapt their behavior based on the context of the application and the user. Fickas et al. [10] define context-aware applications (called environment-direct applications) to be applications that monitor changes in the environment and adapt their operation according to predefined or user-defined guidelines.

Dey et al. [11, 12] claimed it was necessary to give a more generic definition, which is not bound to a specific characteristic (adaptation, reactivity, responsiveness or sensitiveness). Dey's definition states that "*a system is context-aware if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task*". According to this inclusive definition, the only requirement to be a context-aware application is to use context, and features like adaptation, reactivity, responsiveness and sensitiveness to context are not mandatory. This definition covers a broader range of

applications, for instance, applications that do not adapt to context but simply present the context to the user, or applications that do not sense context but make use of context detected by other computing entities. We consider this generic and inclusive definition of context-aware computing as a reference for our work.

To help define the filed of context-aware computing, Dey has given a generic taxonomy to the use of context-awareness: presentation of context and services to a user, automatic execution of a service, and tagging of context to information for later retrieval [12]. *Context Presentation* refers to applications that simply display context information to users, for example, an application displaying GPS coordinates to its user as she moves. *Automatic execution* refers to context-triggered actions, which is the ability to execute or modify a service automatically based on the context. Examples of applications exhibiting this behavior can be a tourist application capable of automatically displaying pages of tourist information as users approach particular attractions [13]. *Contextual tagging* is the ability to associate data with related context. The [14] argues that context can be a powerful cue for recalling past events, for example, when a user is trying to recall a particular document they were editing previously, they may not be able to remember the name of the document, but are more likely to remember that the document was presented at a particular meeting on a particular day with certain colleagues present.

The application of context-awareness has almost unlimited possibilities. To make it more concrete, we list some other examples of context-aware applications in ubiquitous computing environment:

- Location-based applications that allow users to be informed when interested things (ATM, supermarket, restaurants, etc.) or their friends are close to them;
- Bus tracking applications that allow users to keep track of the buses in service, for

example, when a bus is approaching the bus stop and how many seats is available;

- Health-care applications that monitors the health condition of elderly and warns health givers when detecting medical emergencies;
- Emergency applications that warns residents when detecting abnormal condition (smoke, high temperature, window breaking, etc.);
- Context-aware reminder applications that allow users to set reminders to be triggered according to the occurrence of context events;
- Smart mobile phone application that proactively adjust profile (set vibration, change ring volume, redirect call, etc.) to user's situation (meeting, sleeping, working, etc.)
- Appliance control applications that automatically control appliances (adjust temperature and lightening of room, tune volume of A/V devices, etc.) based on residents' preference.

1.3 Challenges in Building Context-Aware Applications

Context-aware computing has been drawing much attention from researchers for nearly a decade. However context-aware applications will never be easily built or widely available to everyday users until following research issues can be fully addressed:

- **How to acquire context?**

  The problem of acquiring context is a prerequisite for any context-aware applications. Context acquisition can be seen as the process that captures the condition of the execution environment, converts this information into machine-understandable form, and creates a digital surrogate for real world contextual environment. Approaches to acquire context are manifold and include employing heterogeneous hardware sensors,

software information providers, and predictive approaches such as inferring user's situation or activity. A necessary property of context acquisition is the reusability of sensors.

- **How to represent context?**

  The problem of representing context is present throughout the entire data flow of context-aware applications, from acquisition, delivery, interpretation, to usage. Raw context data obtained from various sources comes in heterogeneous formats, and applications without prior knowledge of the context representation can't use the data. Hence, an interoperable smart space requires a way to explicitly represent context meanings (or semantics) so that independently developed applications can easily understand them. Previous systems based on ad hoc data models lack in flexibility and expressiveness to represent context, and also fall short in sharing context between different applications as there is no agreement on the representation scheme. We need a systematic approach to model context, allowing better expression of their characteristics and better support for data interoperability.

- **How to deliver context?**

  The delivery of context to applications consists of two problems. The first problem is how to extract a subset of context based on a well-understood information need. For a rich set of context acquired in the contextual environment, it is not practical or desirable to transfer the all context just to use only a small amount of context out of it. Hence query should be the basis for accessing context. The second problem is how to make context available to applications. Some application can use a simple request-response paradigm where an application poses a query and a server generates a finite result set. However other applications may require context be sent continuously whenever it changes. The support for query and notification mechanisms should be

abstract enough to hide low-level processing and communication. In that way application developers can easily implement applications without dealing with underlying details.

- **How to infer higher-level context?**

  Higher-level context such as "What is the user doing?" and "What is the activity in the room?" augments context-aware applications by providing summary descriptions about a user's state and execution surroundings. Usually such abstract information can not be directly recognized by sensors. Instead, higher-level context can be inferred from basic sensed context. For example, an application wants to deduce user's current situation. It could examines whether a given person is currently engaged in a meeting on the basis of location and schedule—if he's in the meeting location and the current time is within the meeting's scheduled interval, he's likely to be at meeting. Providing a flexible support for context inference will make the implementation of context-aware applications much easier.

- **How to support application development?**

  Context-awareness is an enabling technology for ubiquitous computing, so it is commonly required when building ubiquitous computing applications. Suppose that researchers or developers want to prototype a ubiquitous computing application, and that their current interest is in the application's functionality—not the specific distributed architecture and underlying processing of its implementation. To build ubiquitous computing applications efficiently, we need to provide system-level support for context-awareness, including the abstraction of programming model and context data model. Currently many efforts in different aspects of context-awareness are being duplicated, as common problems have to be solved over and over again in each application. Providing a general-purpose system support for *context acquisition*,

context representation, *context delivery*, *context inference* and *context use* will make the process of implementing context-aware applications much simpler.

1.4 Research Contributions

The thesis addresses several important issues in the field of context-aware computing. The main area of work is on context modeling, architectural support, and programming abstractions. The major contributions of the thesis are:

- An ontology-based modeling approach for supporting explicit context representation in term of ontology instances.
- A generic architecture for supporting the acquisition, aggregation, management, inference, and querying of context.
- A set of programming abstractions and their Java implementations for supporting the implementation of sensor wrappers and the prototyping of context-aware applications.
- The implementation of a novel context-aware application (*SituAwarePhone*) that validates and evaluates our architectural support.

The first contribution lies in a novel context modeling approach that allows context to be explicitly described so that independently-developed systems can share context. In our modeling approach, Web Ontology Language (OWL [15]) is adopted as representation language to enable expressive context description and data interoperability of context. The second contribution lies in the generic architecture (Semantic Space) and programming abstractions that provides application designer with a flexible mechanism to build sensor wrappers and prototype context-aware applications. The functionality encapsulated in the architecture handles the common, time-consuming and low-level

details in context-aware computing, allowing designers to concentrate on the application logic while only need a few lines of code to access and use context. One of the novelties is the leverage of Semantic Web technologies (query, inference and knowledge base) to enable the management, query and inference of context. Semantic Space architecture presented in this thesis is designed to be applicable to all ubiquitous computing applications that can usefully benefit from context-awareness. To demonstrate the provided leverage, and to evaluate and validate the principal research contribution of our work, we describe a case study involving the development of a prototypical context-aware application (*SituAwarePhone*) using Semantic Space.

1.5 Outline

The thesis develops in the following way. CHAPTER 2 reviews the related research for this work. This includes an in-depth discussion on existing context-aware applications and demonstrates the potential improvements. CHPATER 3 discusses advantages of the ontology-based approach to modeling context, and presents the design of the context model. The use of this model is also demonstrated by examples. CHAPTER 4 establishes a set of requirements for the architecture designed for context-aware application development. These required features are then incorporated into the Semantic Space, a generic architecture for context-aware applications. CHAPTER 5 presents the implementation of Semantic Space. Semantic Space not only contains implementation of components but also includes a set of APIs for integrating sensors and creating context-aware applications. This chapter also introduces the provided programming abstractions that facilitate context-aware application design. CHAPTER 6 describes the use of Semantic Space in creating a novel context-aware application, namely *SituAwarePhone*, to evaluate the suitability of our architectural solution. Finally, CHAPTER 7 contains a summary and conclusion with suggestions for future research.

CHAPTER 2
BACKGROUND AND RELATED WORK

The design of a new architecture that supports the building and evolution of context-aware applications must naturally leverage off of the work that preceded it. The purpose of this chapter is to describe previous research in the field of context-aware computing. This chapter will focus on both context-aware applications and existing architectural support for building applications. This chapter also gives an introduction to service discovery and Semantic Web technologies that play an important role in our proposed Semantic Space architecture.

2.1 CONTEXT-RELATED PROJECTS

Many projects have investigated context-aware computing; the following sections provide an overview of larger research efforts where context-awareness is a central concern.

2.1.1 Application-Oriented Projects

Many ubiquitous computing projects in the past decade have studied application-oriented approaches to context-aware computing. These projects focused on exploiting interesting application-dependent features of context-awareness.

2.1.1.1 Active Maps

Work on the Active Badge system [16] at Cambridge AT&T Labs is generally considered as the starting point for context-aware computing, to be more specific, location-aware computing. Active Badges are small devices worn by users that transmit a unique

identifier periodically. Sensors deployed on ceilings detect the location of the badge and therefore determine their wearers within the building. Active Badges have been successfully applied by Active Map [3]. The Active Map annotates the graphical floor plans with location information gathered from users and devices (e.g. printers), displaying enhanced map to user so that they can determine to use nearest devices. This project is among the first works to utilized context in ubiquitous computing environment, but the context involved is limited to location information.

2.1.1.2 CyberDesk

The CyberDesk[17] was developed at Georgia Tech's GVU to automatically integrate Web-based systems with context information. CyberDesk uses context related to users' activity to integrate software modules, for example, a user highlighting a particular appointment with a colleague from the diary will be suggested with a number of services. This system is able to interpret the appointment and extract the relevant context that is used by context-aware applications, for example, searching for selected text using a Web search engine, looking up the name in address book, or sending an email to a person. However, the system is very limited in the types of context and does not support multiple simultaneous applications.

2.1.1.3 EasyLiving

The EasyLiving project [18] developed at Microsoft Research focuses on the supporting technologies for intelligent environments, in particular the support for coherent user experience in interacting with a devices based on their presence. EasyLiving encapsulates *hardware device control*, *internal device logic* and *user interface presentation* as basic

component abstractions. All components register with a lookup service and expose a set of attributes so that other components can interact with them. A major focus of this work is location modeling. EasyLiving represents information about the physical relationships between people, devices and places. Its approach uses software components to decouple the sensors from the application, and to provide transparent communications and device discovery mechanisms. As part of the project, an intelligent space prototype has been created to demonstrate several applications including the teleporting of desktops among available displays and dynamic room controllers that provide context-aware access to devices based on location. EasyLiving extensively studied the dynamic discovery of sensors and automatic configuration of devices, but it does not explicitly support the querying and inference of context.

2.1.1.4 CoolTown

CoolTown [19] developed by HP is an Web-based system to support "Web presence" for *people*, *places* and *things* by associating each real world object with a Web resource. Each real-world object is assigned with a Web resource that is automatically correlated with its physical presence. Cooltown primarily supports the display of context to users, for example, a user carrying a PDA electronically picks up URLs for the places traveled through and browses related information from their associated Web pages. The main component of the CoolTown architecture is a *Place Manager* that maintains a directory with the description of the people and objects physically present in a given location. As people and devices move, the state of the directory is updated to represent the current state of the place at any point in time. The CoolTown project successfully created a combined digital and physical Web, however it clearly has limitations. It does not support the structured definition of context. Instead, it allows arbitrary Web description of context.

Context aggregation and querying are also outside the scope of this project.

2.1.2 System-Oriented Projects

While most of application-oriented projects related to context-aware computing relied on *ad hoc* architectures and representations, it was soon recognized that a system-oriented support was key to facilitating practical application development and deployment. In this section we discuss a few projects that specifically address the scalability and flexibility of context-aware applications.

2.1.2.1 Context Toolkit

The Context Toolkit [12] developed at Georgia Tech's GVU aims to separate context acquisition from the actual use of context. The Context Toolkit supports the acquisition and delivery of context using three types of programming abstractions. *Context Widgets* are components that provide applications with access to context sensed from their operating environment. They free applications from the context acquisition process by hiding the complexity of the sensors' details. Each widget encapsulates state and a set of event callbacks. The state is comprised of context that applications can access through subscription. Callbacks represent the types of events that the widget can use to notify subscribing applications. The Widget also maintains contextual state allowing external components to retrieve context information. *Context Aggregators* are used to collect the entire context about a particular entity such as a user or a room. It is responsible for subscribing to all widgets of interest and acts as a proxy to the application, collecting information for that particularly entity. *Context Interpreters* are responsible for the interpretation of context information. They transform between different representations formats or fuse different context into new high-level context.

As the seminal work in providing architectural support for context-aware computing, the Context Toolkit is a starting point for developers to build context-aware applications in a systematic way. However, Context Toolkit applies a simple context specification mechanism based on attribute-value pairs, which falls short in expressiveness of representing features of context (rich typing of context, relation between different context, etc.). While it is convenient for applications to select desired context using simple attribute-value matching, this approach offers little control because of the limited expressiveness in selective access. Moreover, the implementation of interpreters requires low-level programming when the application needs to infer a new type of high-level context.

2.1.2.2 Context Fabric

The Context Fabric project [20] at UC Berkeley developed a service-oriented context-aware infrastructure. Context Fabric provides two fundamental built-in services, namely event service and query service, to support the acquisition and retrieval of context. Context Fabric uses an entity-relationship based context model to represent four kinds of context-related concepts: entities, attributes, relationships, and aggregates. Context about each kind of entities are assigned network-addressable logical storage units called *infospaces* that can be queried by applications. The Context Fabric allows application to specify a high-level query, and based on the type of requested context, it automatically constructs a data-flow path by selecting operators from a repository. As context is encoded in XML, Context Fabric employs XPath as the query language.

Context Fabric's entity-relationship based context modeling approach takes a step forward than Context Toolkit in supporting more expressive representation of context. While it supports automatic path composition of context processing operators, this

approach also restrict the expressiveness of context query due to the limited expressiveness of type matching. Moreover, Context Fabric lacks in the support for the inference of implicit context.

2.1.2.3 Solar

The Solar project [21] of Dartmouth College developed a graph-based programming abstraction for context aggregation and dissemination. A Solar architecture has two kinds of clients: *Sensors* as data sources and *applications* as data sinks. A sensor may publish a data stream, by pushing data items called events into Solar. Some sensors also may have a pull interface that allows users to query its current state. Applications ask Solar to find specified sensors and to execute application-supplied data-fusion operators to compute context. An operator is an independent data processing module that takes one or more data sources as input and acts as another data source. A number of event processing and routing mechanisms are designed to avoid redundant computation at intermediate nodes for aggregation and interpretation, and reduce data transmission in large-scale context-aware system.

Unlike the centralized design of *Context Aggregators* in the Context Toolkit, Solar support the aggregation and dissemination of context using distributed event graph. This approach eliminates the dependence on central control components, thereby helping to achieve a high level of scalability in gathering and delivering context. Similar to the specification language of the Context Toolkit, Solar also uses an attribute-value based context description scheme, leading to the limited expressiveness in context representation. In addition, the need for an *application* to poll a *sensor* to determine when an interesting change has occurred is an unnecessary burden for application developers.

2.1.2.4 TEA

The Technology for Enabling Awareness[22] approaches to systematically supporting context-awareness involving the conversion from low-level sensor data to abstract contextual information, for example, transforming sensors values such as "number of user >2", "light = 90%", "noise = 75%" into situation such as "at meeting". This is achieved through the *Layered Perception Architecture*, which consists of three layers: *sensor layer*, *cue layer* and *context layer*. Output from sensors (layer 1) is regarded as low level and requires transformations or filters before being understandable to applications. Transformation of raw sensor data is preformed in the cues layer (layer 2), where the output is a better interpretation of the data. The context layer (layer 3) involves the mapping of cues to abstract situation, which forms the focus of the TEA approach. The context layer enables applications to utilize the abstract context in order to adapt their behaviors.

TEA's cues and situation abstractors provide a separation of concerns between how context is acquired and how it is used by applications. However, there is limited support for specifying what context an abstractor or application is interested in. Also, there is almost no support for the explicit representation and expressive querying of context.

2.1.3 Discussion of Related Work

In the previous sections, a range of characterizations and definitions for context-aware systems has been surveyed and analyzed. We categorize context-related research projects into application-oriented projects and system-oriented projects. The problems of application-oriented projects lie in *ad hoc* architectures and limited support for flexibility, scalability and interoperability. Related research suggests software architectures and programming abstractions to ease the use of context when developing context-aware

applications.

Most recent context-related architectures have realized the importance of providing abstract *context acquisition* mechanisms that is, decoupling low-level sensing and actual context use. A common method is to provide developers with a set of reusable components (in form of application programming interfaces) that support their implementation tasks in sensing and handling context. Our work resembles these projects in providing a level of abstraction between sensors and context-aware applications (CHAPTER 4).

Current approaches to *context representation* generally lack of expressive power and formal foundation. Most early projects uses attribute-value pair (AVP) based data model to describe context, consequently, simple attribute matching is the only choice to the *context query* support. Even though Context Fabric uses a more expressive entity-relationship (E-R) model to model context and XPath to query context, it still has problem in supporting interoperability and data sharing across independently-developed systems. Therefore the potential improvement here is to develop an expressive context model that can support flexible query and data interoperability of context. To advance this matter an ontology-based context model was developed and is presented in CHAPTER 3.

The ultimate goal of using context is to make available to the system a representation of the real-world situation. The gap between sensor data and higher-level situation has to be filled by *context inference* mechanisms. Related research realizes context inference using technologies ranging from simple format mapping to machine learning based interpretation. However, these projects use hard coding approaches, that is, they do not provide any generic mechanism for writing rules about context or inferring higher-level context in a structured format. In Semantic Space, the use of ontology to handle context lends itself directly to using logic rules to automate inference process. Using rule-based

inference, application developers can script rules, but write application code, to deduce higher-level context.

Ubiquitous computing environment is comprised of large amounts of frequently changing context; hence continuous update of new context to applications is especially useful for *context delivery*. Besides simple query-response mechanisms, some projects applied subscription-based context delivery protocols. However, none of previous systems provides support for both the continuous notification and expressive query of context. We advance this matter by providing continuous query support (CHAPTER 4). Applications can register very expressive queries that specify their interests over changing context; the query service continuously filters and synthesizes incoming context from sources, and delivers streaming results to the appropriate applications.

2.2 RESOURCE DISCOVERY IN UBIQUITOUS COMPUTING ENVIRONMENT

Networked devices, ranging from tiny sensors to powerful devices provide a variety of contextual information. It is very important to dynamically locate and configure these context providing devices (or rather their software wrappers). Current research in resource (or service) discovery protocols make devices and services hosted by them easier to use; they facilitate interaction between computing entities, with an aim to approach zero-configuration overhead and therefore free users from administrative and configuration work. General-purpose support for resource discovery provides a basis for the realization of the dynamic discovery of context providers. We will now briefly review some of the general-purpose discovery protocols: SLP, Jini and UPnP. The following section looks at each of them in turn.

- **Service Location Protocol**

Service Location Protocol (SLP) [23] is an IETF standard for enabling IP-based applications to automatically discover the location of a service. The SLP defines three "agents": User Agents (UA) that perform discovery on behalf of client software, Service Agents (SA) that advertise the location and attributes on behalf of services, and Directory Agents (DA) that store information about the services announced in the network. SLP has two different modes of operation: when a DA is present, it collects all service information advertised by SAs and the UAs unicast their requests to the DA, and when there is no DA, the UAs repeatedly multicast the request they would have unicast to a DA. SAs listen for these multicast requests and unicast their responses to the UA.

- **Jini**

  Jini [24] is a service discovery protocol developed by Sun Microsystems. Its goal is to enable truly distributed computing by representing hardware and software as Java objects that can form themselves into communities, allowing objects to access services on a network in a flexible way. Service discovery in Jini is based on a directory service, similar to the Directory Agent in SLP, named the Jini Lookup Service (JLS). JLS is necessary to the functioning of Jini, and clients should always discover services using it and can not access services directly.

- **Universal Plug and Play**

  Universal Plug and Play(UPnP) is an architecture for peer-to-peer network connectivity of devices and computers. It's introduced as an extension to the Intel's plug and play peripheral model In UPnP, a device can dynamically join a network, obtain an IP address, convey its capabilities upon request, and learn about the presence and capabilities of other devices. Finally, a device can leave a network smoothly and automatically without leaving any unwanted state behind [25].

Simple Service Discovery Protocol (SSDP) [26] was created as a lightweight discovery protocol for UPnP, and it defines a minimal protocol for multicast-based discovery. SSDP can work with or without its central directory service. When a service wants to join the network, first it sends an announcement message to notify its presence to the rest of the devices. This announcement may be sent by multicast, so all other clients (each with an UPnP control point) will see it. After the client has retrieved a description of the device, it can send control message to a device's service using remote procedure call. Control messages are expressed in XML using the Simple Object Access Protocol (SOAP).

Unlike Jini and SLP which rely on centralized directory mechanism, UPnP does not require the presence of a central point. The support for peer-to-peer connectivity is desired in ubiquitous computing environment. From a system implementation perspective, the programming of UPnP services is language independent. Therefore we apply UPnP in our architecture to support the automatic discovery of context providing components.

2.3 SEMANTIC WEB OVERVIEW

Today's World Wide Web infrastructure is moving towards the Semantic Web vision that aims to make Web resources more readily accessible to automated processes by adding metadata annotations that describe their content [27]. In this way, the Semantic Web can allow both humans and computers to make use of data in ways that previously haven't been possible.

There have been a wide range of applications growing from Semantic Web technologies. According to [28], current Semantic Web-enabled software applications include:

- *Content creation* that allows authors to connect metadata (subject, creator, location,

language, copyright status, or any other terms) with documents, making the new enhanced documents searchable;

- *Content management* that allow large-scale Web sites to be managed dynamically according to content categories customized for the site managers;

- *Resource description and discovery* that allow organizations to integrate enterprise applications, publishing and subscriptions using flexible semantic models;

- *Data reuse* that standardizes RDF and OWL based data models, allowing data reuse from diverse sources.

The potential of Semantic Web can never be limited to the above mentioned applications. In this thesis, we exploit the use of Semantic Web technologies in a new area – context-aware computing. In this thesis, we will present a software architecture that makes use of Semantic Web technologies to support explicit representation, expressive querying, and rule-based inference of context in ubiquitous computing environment. Before we start to present the details of our system, we give an overview of the Semantic Web technologies.

2.3.1 Semantic Web Standards

The essence of the Semantic Web is a set of standards for the exchange of descriptions of Web entities and their relationships, together with a set of supporting tools that provide a federated ontology-based approach to knowledge management. To fully realize Semantic Web, a number of standards (XML, RDF, RDFS, DAML, OIL, OWL, etc.) for describing and exchanging machine-understandable content have been developed. Among various standards, the W3C Consortium announced final approval of two key standards: the Resource Description Framework (RDF) and the Web Ontology Language (OWL) [28]. RDF and OWL provide a standardized framework for asset management, enterprise

integration and the sharing and reuse of data on the Web.

- **XML**

  At the foundation, XML provides a set of rules for creating vocabularies that brings structure to both documents and data on the Web. XML also gives clear rules for syntax when XML Schemas serve as a method for composing XML vocabularies. XML is a powerful, flexible surface syntax for structured documents; however it does not impose semantic constraints on the meaning of documents.

- **RDF**

  RDF [29] is a standard a way for making simple descriptions about Web content. Built based on XML which is used for syntax, RDF contains a clear set of rules for providing simple descriptive information at semantic level. RDF Schema (RDFS) then provides a way to combine multiple RDF descriptions into a single vocabulary.

  RDF is based on a concrete formal model that uses directed graphs for representing the semantics of metadata. The core of every RDF expression is in a *(subject, predicate, object)* triple. Every triple consist of the subject (resource being described), predicate (named property), and object (the value of this property). Resources and predicates are represented by URIs. This abstract syntax of RDF is serialized using several alternative concrete syntaxes, like RDF/XML [30], N3[31], N-Triple[32].

- **OWL**

  OWL [15] by its nature is to develop domain-specific vocabularies through the use of ontology. Within the domain of knowledge representation, the term *ontology* refers to the formal, explicit description of concepts, which are often conceived as a set of entities, relations, instances, functions, and axioms. Ontology can encode knowledge in a domain and also knowledge that spans different domains, thereby enabling a level of knowledge reuse.

Among Semantic Web standards, OWL provides a language for formally defining structured, Web-based ontologies which delivers richer integration and interoperability of data among descriptive communities. OWL builds on RDF and RDFS and adds formal vocabulary for describing properties and classes, for example, whether a class is equivalent to or disjoints with another class, whether a property is transitive, symmetric, functional or inverse to another property.

OWL provides three increasingly expressive sublanguages: *OWL Lite*, *OWL DL* and *OWL Full*. OWL Lite supports classification hierarchy and simple constraints. OWL DL supports maximum expressiveness while retaining computational completeness (all conclusions are guaranteed to be computable) and decidability (all computations will finish in finite time). OWL DL includes all OWL language constructs, but they can be used only under certain restrictions (for example, while a class may be a subclass of many classes, a class cannot be an instance of another class). OWL DL is so named due to its correspondence with *description logics*, a field of research that has studied the logics that form the formal foundation of OWL. OWL Full is designed for maximum expressiveness and the syntactic freedom of RDF with no computational guarantees. OWL Full allows an ontology to augment the meaning of the pre-defined (RDF or OWL) vocabulary. Reasoning mechanism will not be able to support complete reasoning for all features of OWL Full.

2.3.2 Jena2 Semantic Web Framework

Jena2 is a software framework developed by Hewlett-Packard Labs Bristol for building Semantic Web applications [33]. It is an implementation for W3C's Semantic Web recommendation, provides a programmatic environment for RDF and OWL, including the support for expressive RDF query, generic rule-based inference, and scalable persistent storage.

The heart of the Semantic Web recommendations is the RDF *Graph* [29], as a universal data structure that consists of a set of triples. Jena similarly has the *Graph* as its core abstraction for manipulating RDF and OWL. The key supports of Jena2 include:

- **RDF and OWL API**

  Jena2 supports flexible presentations of RDF graphs to application programmers. This allows easy access to and manipulation of data in RDF graphs, enabling programmers to navigate the triple structure in an abstract way. Particularly, the *Model* API presents the graph using the terms and concepts from the RDF recommendations, and the *Ontology* API presents the graph using concepts from OWL and RDFS.

- **RDQL Query Language**

  Jena2 supports RDQL (RDF Data Query Language [34]), the *de facto* reference implementation of RDF query language. This allows programmers to define declarative, SQL-styled query statements to extract information from a RDF graph.

  An RDQL consists of a graph pattern expressed as a list of *triple patterns*. Each triple pattern is comprised of named variables and RDF values(URIs or literals). An RDQL query can additionally have a set of constraints on the values of those variables, and a list of the variables required in the answer set. Figure 2 shows a self-explanatory RDQL query asking "all the rooms with temperature below 20 degree". The detailed grammar of RDQL is listed in APPENDIX A.

```
SELECT ?room
WHERE  (?room, rdf:type, ss:Room),
       (?room, ss:hasTemperature, ?temperature)
AND    (?temperature < 20)
USING  ss FOR <http://www.i2r.org.sg/semanticspace#>,
       rdf FOR <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

Figure 1: A sample RDQL query specification

- **Scalable RDF Persistence**

  The Jena2 persistence subsystem implements an extension to the *Model* API that

provides scalable persistence for RDF models through use of a back-end database. It also supports a fast-path capability for RDQL queries that dynamically generates SQL queries within a relational database.

- **Rule-Based Inference**

  The Jena2 reasoner subsystem includes a generic rule based inference API together with configured rule sets for RDFS and for the OWL Lite. These reasoners can be used to construct inference models which show the RDF statements entailed by the data being reasoned over.

  This general-purpose inference engine supports *forward chaining* reasoning that allows developers to define their own horn-logic rules. The rule specification comprises a list of *antecedents* (body) and a list of *consequents* (head). Each element in the head or body is a set of conjunctive *triple patterns*, that is, a triple of nodes to represent constant URI, literal graph nodes, as well as variables or wildcards. For example, Figure 2 shows a rule expressing the fact that, given "which room is the user in?" and "which building is the room a part of?" we can derive "which building is the user in?". The detailed rule syntax of Jena2 inference engine is given in APPENDIX B.

```
type(?user,User) ∧ type(?room, Room) ∧ type(?building,Building) ∧
locatedInRoom(?user,?room) ∧ isPartOf(?room, ?building)
=>locatedInBuilding(?user, ?building)
```

Figure 2: A sample inference rule in Jena2 rule syntax

2.4 THE ROLE OF SEMANTIC WEB IN CONTEXT-AWARE COMPUTING

Semantic Web enables both humans and computers to exchange and process data in ways that previously haven't been possible. While the Semantic Web standards and the supporting tools are originally designed for Web-based applications, we show that it is well suited to many requirements of ubiquitous computing environments.

From a representation point of view, using ontologies to model context in ubiquitous computing environments offers several advantages. First, OWL is an expressive language that can be used to describe rich features of context (e.g., various types of entities, properties of entities, relationships between entities, and constraints on this information). By allowing ubiquitous computing entities to share a common understanding of the structure of context, OWL ontologies can enable independent-developed applications to share and interpret contexts based on their semantics, even when these applications don't have predefined agreements on how they should interoperate. Additionally, ontologies' hierarchical structure lets application developers reuse domain ontologies (for example, of people, devices, and calendar) in describing context and build a practical context model without starting from scratch. Further more, because context described in ontologies have explicit semantic representations, ontology-described context lends itself directly to the use of knowledge base to support expressive query and automated inference.

When context is represented in OWL/RDF, we can leverage Semantic Web tools to facilitate different management and processing tasks for context-aware applications. By applying Semantic Web inference engine, context-aware applications can use specific logic rules to deduce implicit, higher-level context from explicit, lower-level context. Taking this approach, developers can customize context inference simply by defining rules. Further more, Semantic Web query service can be leveraged to facilitate expressive context query, allowing applications to access context through the use of declarative queries.

Another important role of Semantic Web lies in the context acquisition. While Semantic Web is the most effective medium for people to get information, it also can be seen as a rich context source for all kinds of context-aware applications. As the Semantic Web

matures, the future Web will be populated with vast amount of explicit represented information that will be invaluable for acquiring context. Given the interoperability of ontologies, Semantic Web annotations such as street directories, restaurant menus and personal profiles can be incorporated into ubiquitous computing applications as useful context.

The application of Semantic Web can never be limited to Web-based systems, it also has great potential in ubiquitous computing, specifically context-aware computing. In this paper, we will present our approach to exploit the use of Semantic Web technologies in building a software architecture for context-aware applications.

2.5 SUMMARY

This CHAPTER has presented a comprehensive survey of research projects related to context-aware computing. Application-oriented projects tend to offer *ad hoc*, application-specific mechanisms to application design and, in general, often neglect the support for flexibility, scalability and interoperability of context-aware systems. System-oriented projects specifically address the architectural solution for context-aware applications. We have learned quite a bit from previous work. As we will see in CHAPTER 4, based on we extract from related work a set of design requirements that would be useful across all context-aware applications. They include context capture, explicit representation, context inference, expressive query, continuous delivery, dynamic discover and programming abstraction.

Apart from related work, this chapter also introduced background knowledge about Semantic Web, which is a part of the enabling technologies for context-awareness in our approach. Following chapter will present the use of Semantic Web standards to design an ontology-based context model.

CHAPTER 3
CONTEXT MODEL

Context representation is an important part of ubiquitous computing environment, and an appropriate context model is the basis for context representation. There are different kinds of context with different characteristics and complex interrelationships. A generic architecture for context-awareness should use a systematic approach for context model that expressively captures the rich features of context. Furthermore, the determination of the context model should consider data interoperability issues, since the context-aware architecture is expected to support independent ubiquitous computing applications. In addition, the context model should be extensible to allow adding new kinds of context that have not been anticipated during the architecture design. In this chapter, we present an ontology-based context model that can fulfill the above requirements.

3.1 ONTOLOGY-BASED APPROACH TO CONTEXT MODELING

Within the domain of knowledge representation, the term ontology refers to the formal and explicit description of domain concepts, which are often conceived as a set of entities, relations, instances, functions, and axioms. Among Semantic Web standards, OWL is used to define and instantiate ontologies that let distributed computing entities exchange and process information based on a common vocabulary. We observed that using ontologies to model contexts in pervasive computing environments offers several advantages:

- OWL is a structured representation language that is sufficiently expressive to describe rich features of context, including types of contextual entities, properties of entities, relationships between entities, and constraints on entities and relationships.

- By allowing ubiquitous computing entities to share a common understanding of context structure, OWL ontologies enable applications to process contexts based on their semantics.

- Ontologies' hierarchical structure lets developers reuse domain ontologies (for example, of people, devices, and activities) in describing context and build a practical context model without starting from scratch. The ontology approach is extensible to add new concepts about context in a hierarchical manner.

- Because context described in ontologies have explicit semantic representations, Semantic Web tools such as declarative query, logic inference, and knowledge base can be directly leveraged. Incorporating these tools into context-aware architecture facilitates context management and processing.

3.2 PRIMITIVE CONTEXTUAL ENTITIES

With the increase of complexity in ubiquitous computing environment, it is useful to identify general context that is applicable to context-aware applications. Taking an object-oriented modeling approach, we have identified three classes of physical entities (*user*, *location*, *computing entity*) and one class of conceptual entity (*activity*) that play an critical role in characterizing a ubiquitous computing environment. Properties of these entities, as well as the relationships between them, form the skeleton of a general contextual environment. Furthermore, primitive contextual entities can provide indices into associated context, for example, given a location, we can acquire related context such as the temperature and noise level of it, people and activity inside it, and so on. The identification of primary contextual entities forms the basis for our ontology-based context model.

To model primitive context, the context model should be able to represent *intrinsic* context, that is, attribute of a contextual entity (e.g., a user's name) as well as *extrinsic* context, which is a relationship between entities (e.g., a user is in a location). Additionally, the context model needs to represent *static* pieces of context (e.g., an email address of a user) that don't change or change slowly, as well as *dynamic* context (e.g., a user's location) which changes often. The desired features of context modeling are summarized in Table 1.

Table 1: Features of Primitive Contextual Entities

	Static / Intrinsic	**Dynamic / Intrinsic**	**Static / Extrinsic**	**Dynamic/ Extrinsic**
User	name, SSN, address, e-mail, DoB, title	heart beat rate, blood pressure, temperature, situation(e.g. meeting, sleeping), mood	a user is a friend of another user, a user owns a device, a user is licensed to use a software	a user is in a location, a user is in the same location with another user, a user is involved in an activity
Location	name, size coordinates, type(e.g. district , building, room),	temperature, light, noise, door status (open, closed), number of users in the room	a room is a part of building, a location is disjoint with another location	a room is occupied by user(s), a room is hosting activity(s), a location is equipped with device(s)
Comp. Entity	name, mode, product sn, type, capability attributes	status(e.g. whether a PowerPoint is running, whether a mobile phone is being used)	a device is own by a user, a software is licensed to a user	a device is within a room, a device is being used in an activity
Activity	name, time, type(e.g. meeting, sleeping, taking call, discussing)	status(e.g. obsolete activity, ongoing activity, future activity)	attendees, location, involved computing entity(e.g. projector, PowerPoint)	attendee(s) currently present at the activity

3.3 DESIGNING THE CONTEXT MODEL

Ubiquitous computing environments cover a range of environment types such as homes, offices, workplaces, classrooms, and vehicles. Instead of trying to completely model all types of context in different kinds of contextual environments with varying features, we

define an *Upper-Level Context Ontology* (*ULCO*) to provide a set of basic concepts common across different environment types. Among different types of context, we choose to model primitive contextual entities and a set of their sub-classes in ULCO(Figure 3).

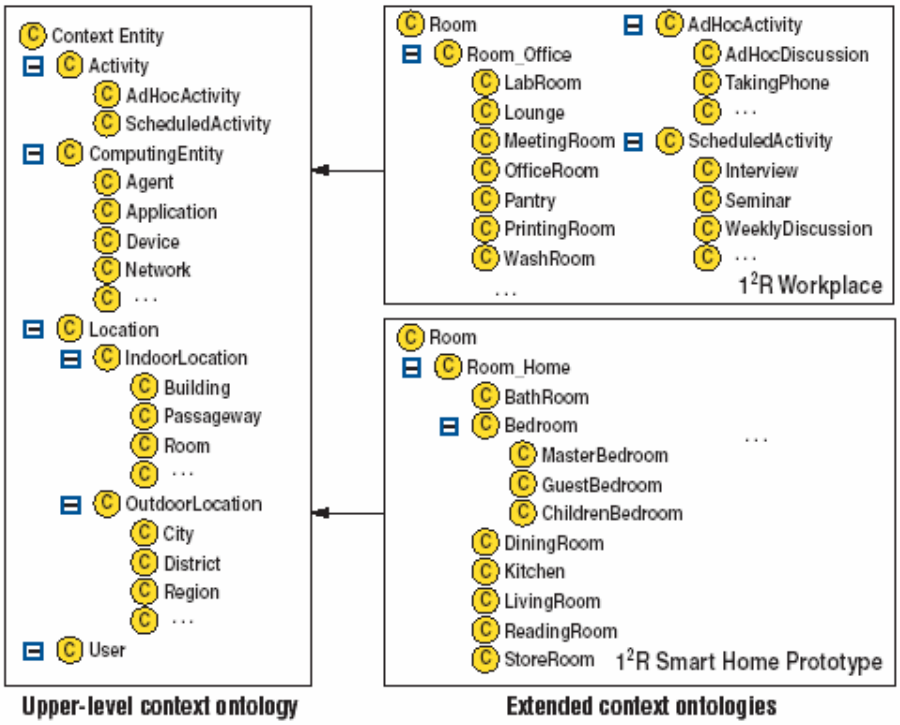

Figure 3: Upper-level context ontology and extended context ontologies

We use OWL to express ontologies of the context model. Description of context is structured `owl:Class`, each of which describing a physical or conceptual object in the contextual environment. Each class is associated with its attributes (represented in `owl:DatatypeProperty`) and relationships with other classes (represented in `owl:ObjectProperty`) to describe both intrinsic and extrinsic context. The built-in property `rdfs:subClassOf` creates a easy way for developers to add new context hierarchically. Following shows the partial OWL/XML representation of UCLO.

```
<!-- name space and ontology definition-->
<rdf:RDF
    xmlns=”http://www.i2r.org.sg/semanticspace#”
    xmlns:rdf=”http://www.w3.org/1999/02/22-rdf-syntax-ns#”
    xmlns:rdfs=”http://www.w3.org/2000/01/rdf-schema#”
    xmlns:owl=”http://www.w3.org/2002/07/owl#”
    xml:base="http://www.i2r.org.sg/semanticspace#">
<owl:Ontology
 rdf:about=”http://www.i2r.org.sg/semanticspace#”
 dc:title="Semantic Space's Upper-Level Context Ontology "/>
<!-- partial definition of classes -->
<owl:Class rdf:ID="ContextEntity"/>
<owl:Class rdf:ID="Location">
   <rdfs:subClassOf rdf:resource="#ContextEntity"/>
</owl:Class>
<owl:Class rdf:ID="IndoorLocation">
  <rdfs:subClassOf rdf:resource="#Location"/>
</owl:Class>
<owl:Class rdf:ID="OutdoorLocation">
  <rdfs:subClassOf rdf:resource="#Location"/>
  <owl:disjointWith rdf:resource="#IndoorLocation"/>
</owl:Class>
...
<owl:Class rdf:ID="User">
   <rdfs:subClassOf rdf:resource="#ContextEntity"/>
</owl:Class>
<!-- partial definition of properties-->
<owl:ObjectProperty rdf:ID="locatedIn">
   <rdfs:domain>
        <owl:Class>
            <owl:unionOf rdf:parseType="Collection">
                <owl:Class rdf:about="#User"/>
                <owl:Class rdf:about="#Activity"/>
                <owl:Class rdf:about="#Device"/>
            </owl:unionOf>
        </owl:Class>
   </rdfs:domain>
   <rdfs:range rdf:resource="#Location"/>
   <owl:inverseOf rdf:resource="#contains"/>
</owl:ObjectProperty>
<owl:DatatypeProperty rdf:ID="SSN">
   rdf:type rdf:resource=
   "http://www.w3.org/2002/07/owl#InverseFunctionalProperty"/>
   <rdfs:domain rdf:resource="#User">
   <rdfs:range rdf:resource=
   "http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>
</rdf:RDF>
```

Figure 4: A partial XML representation of UCLO

OWL can describe concepts that are more expressive than entity-relationship models – OWL can include formal restriction and characteristics on classes and properties, enabling the expression of more features about context. In the XML segment described in Figure 4, we define the class `IndoorLocation` and the class `OutdoorLocation` to be disjoint to guarantee that an indoor place cannot simultaneously be an instance of an

outdoor location (`line 21`). We also define the attribute `SSN` of the class `User` to be `InverseFunctionalProperty` so that it can be used as a unique key for identification of a user (`line 43`).

To let developers customize the context model for a particular ubiquitous computing environment, ULCO allows the definition of new concepts in term of subclasses to complement the upper-level classes. A new application that needs additional classes can obtain them by inheritance from the ULCO classes, forming an *Extended Context Ontology,* as Figure 3 shows. This lets application developers easily build detailed context models for a new contextual environment. Moreover, by providing shared terms and definitions for context, ULCO supports better interoperability between extended ontologies.

Knowledge reuse is one important advantage of ontologies. When defining context ontologies, we integrated consensus domain ontologies such as friend-of-a-friend (FOAF, `xmlns.com/foaf/0.1`), RCAL Calendar (`www.daml.ri.cmu.edu/Cal`), and FIPA Device Ontology (`www.fipa.org/specs/fipa00091/XC00091D.html`) into ULCO to model context about *user*, *activity*, and *device* respectively. These well-defined ontologies provide generic vocabularies that suit context ontologies' requirements - we need only add additional properties useful to ubiquitous computing environments. For example, FOAF defines simple relationship between people (that is, `friendOf`), but we extend it to support richer properties such as `supervisorOf`, `studentOf`, and `colleagueOf` to describe specific information.

3.4 APPLYING THE CONTEXT MODEL

The nature of context representation is to create a digital surrogate for real world contextual environment. Using the context model, we represent context as ontology

instances and associated properties (context markups), which applications can semantically understand and process.

Context often originates from diverse sources, leading to dissimilar approaches to generating context markups. Static context such as a person's name and scheduled seminar time have relatively slow change rates, and users often supply this information. Users also usually generate markups of static context. For example, we use a Web-based application that let users create online profiles based on the ontology class `User` defined in the context model. Figure 5 shows the context markup that describes static context about user `XiaohangWang`. (Throughout the thesis, we assume `www.i2r.org.sg/semanticspace#` as the default base namespace.)

```
<User rdf:about="XiaohangWang">
  <name>Xiaohang Wang</name>
  <mbox>xwang@i2r.org.sg</mbox>
  <homepage rdf:resource="www.i2r.org.sg/~xiaohangWang"/>
  <office rdf:resource="#Room209"/>
  <mobilePhone>6789</mobilePhone>
  <supervisorOf rdf:resource=#DaqingZhang"/>
  <supervisorOf rdf:resource=#JinsongDong"/>
  <!-–More properties not shown in this example–->
</User>
```

Figure 5: XML representation of the instance describing a `User`

Each OWL instance has a unique URI, and context markups can link to external definitions that are not directly included through these URIs. For example, the URI `www.i2r.a-star.edu.sg/SemanticSpace#XiaohangWang` refers to the user we just defined, and the URI `www.i2r.a-star.edu.sg/SemanticSpace#Room209` refers to a specific room defined elsewhere.

Some static context is not directly inputted by users, but extracted from software resources. For example, to generate context markup for scheduled activities, we make use of Retsina Semantic Web Calendar Agent (`http://www.daml.ri.cmu.edu/Cal`), which automatically transforms event schedules from Microsoft's Outlook2000 into instances of

ontology class `ScheduledActivity`, as Figure 6.

```
<ScheduledActivity rdf:id=" http://www.i2r.org.sg/event#040809a">
  <title>Semantic Space Introduction</title>
  <description>An infrastructure for context-awareness</description>
  <startTime>20040809T090000</startTime>
  <endTime>20040809T110000</endTime>
  <hasLocation rdf:resource="#MeetingRoom609"/>
  <attendee rdf:resource="#XiaohangWang"/>
  <attendee rdf:resource="#YuhuiWu"/>
  <!--More attendees are not shown in this example-->
</ScheduledActivity>
```

Figure 6: XML representation of the instance describing a `ScheduledActivity`

Hardware and software information sources usually provide dynamic context such as user's location, current time, noise, or temperature. For dynamic context that change frequently, automated programs are involved in marking up this information. Consider the RFID (Radio Frequency Identification) location system that we developed to tracks users' indoor location by detecting the presence of body-worn tags. When the user `XiaohangWang` enters `Room209`, the RFID sensor could detect his presence and automatically compose the following context markup.

```
<User rdf:about="#XiaohangWang">
  <locatedInRoom rdf:about="#Room209"/>
</User>
```

3.5 SUMMARY

This chapter argued the advantages of ontological approach to context modeling, and presented an ontology-based context model that essentially enables the formal, explicit representation of context. To achieve better extensibility and interoperability, the context model is designed into a two-level hierarchy. Upper-level context ontology provides a standard vocabulary modeling primitive contextual entities that are common across all context-aware applications. Developer could further extend the upper-level ontology with additional classes and properties, forming an extended context ontology to detail a

specific contextual environment. In Semantic Space, sensor input and other contextual data are transformed into semantic markups based on the context ontology before flowing to the context infrastructure and applications. In this way, context can be semantically exchanged between independently-developed systems. Furthermore, this approach creates an opportunity for us to incorporate Semantic Web technologies, such as query, inference and knowledge base, into the context architecture to support processing and management tasks.

CHAPTER 4
AN ARCHITECTURE FOR CONTEXT-AWARE APPLICATIONS

An architectural support for context-aware applications should provide generic functionality to address the necessary requirements of context-aware computing. The goal of this chapter is to identify these essential requirements and presents a generic context architecture that both provides necessary features and programming abstraction. This architecture incorporates technologies from Semantic Web community in efficiently supporting various tasks in context-awareness. More precisely, the context infrastructure is designed to provide application developers with a convenient mechanism for supporting context-awareness through an appropriate set of abstractions for integrating sensors and building applications.

4.1 REQUIREMENTS

This section uses observations obtained from context-related projects and our context-aware connected home project[35] to establish a generic set of requirements for supporting context-aware applications within ubiquitous computing environment. The objective of this analysis is to identify the core architectural features required by a context architecture to better facilitate context-awareness.

- **R1: Context Capture**

  The first step to use context is to capture context from the contextual environment. This introduces a requirement to support a wider variety of context sensed from both physical and virtual world. Physical context includes anything that can be sensed by

hardware devices such as RF devices or environment sensors. Virtual context refers to context obtained through the use of software components, for example, monitoring keyboard activity, device status or processor load.

- **R2: Explicit Representation**

  Raw context obtained from sources comes in heterogeneous formats, and applications without prior knowledge of its representation can't use this data. Therefore, an interoperable system support for context-aware applications requires a way to explicitly represent context's meanings (or semantics) so that independently developed applications can easily understand them.

- **R3: Context Inference**

  For applications to successfully utilize context in a meaningful way, inference of higher-level context is required. Higher-level context (e.g. what is the user doing? what is the activity in the room?) augments context-aware applications by providing summary descriptions about a user's state and surroundings. As sensor devices can't directly recognize such context, the context architecture should provide a support for applications to infer this information from basic sensed context.

- **R4: Expressive Query**

  While the contextual environments may maintain a large amount of context, a particular context-aware application only needs to selectively access a subset of the context. The context architecture should be able to answer expressive queries that can well specify application's context need - for example, "who is in the room with the user?", "when will the meeting the user is attending end?". Query involves application developers defining their context need using declarative query specification.

- **R5: Continuous Delivery**

In a highly dynamic ubiquitous computing environment, the delivery of context based on request-response mode is not able to continuously feed applications with up-to-date context; application developers have to handle this problem by polling context sources in an *ad hoc* manner. To advance this matter, the context architecture should support continuous context delivery mechanism in which applications register query specifications, and a continuous query engine filters the query result to deliver streaming context to applications.

- **R6: Dynamic Discovery**

  The dynamism of ubiquitous computing environment influence the need to support discovery and configuration of context sources (or their software wrappers). When a new context source joins the contextual environment, the context architecture and applications should be able to locate and access it, and when the context source leaves the contextual environment, applications should be aware of its unavailability to avoid stale information. A dynamic discovery mechanism should include support for the discovery life cycle presented in CHAPTER 2, including advertisement and discovery.

- **R7: Persistent Storage**

  Exploiting context through persistence involves useful information being gathered from the contextual environment and stored for later retrieval. Persistence can be exploited by applications themselves to infer new knowledge or establish trends. Consider a tour guide application, location of users can be persistently stored and later utilized to determine the popular sites within the city and information that is often requested at those sites.

- **R8: Programming Abstraction**

  Context-aware applications must be able to utilize a wide range of computing devices, communications and sensing technologies and implement their own functionalities

irrespective of the underlying architecture. To ease the development of, a system support for context-awareness needs a set of programming abstractions to decouple enabling mechanisms with application's functionality. These programming abstractions allow developers to implement context-aware applications and to integrate sensor components in a simple way without dealing with low-level processing.

4.2 ARCHITECTURE OVERVIEW

Based on the requirements specified in the previous section, the remainder of this chapter introduces the key components of Semantic Space architecture before detailing the design of a prototype implementation. Semantic Space provides a generic architectural support that allows context to be represented as semantic markups that applications can easily interpret. It enables applications to access context using queries, and support the inference of higher-level context from basic context using logic rules. One such architecture is maintained in each smart space (or contextual environment), which is often bound by the physical space in a non-restricted way. For example, two or more rooms may be joined to form a smart space, or a single room may be split into multiple smart spaces.

Semantic Space architecture consists of collaborating components, as shown in Figure 7 *Context Wrappers* obtain raw data from software and hardware sensors, transform this information into semantic representation and publish it for other components to access. *Context Aggregator* discovers distributed wrappers, gathers context from them and updates *Context Knowledge Base* (*KB*) asynchronously. Context KB dynamically links context into a single coherent data model, and provides interfaces for context reasoner and context query engine to manipulate stored context. *Context Reasoner* is a rule-based inference engine that can infer higher-level context from stored context. *Context Query Engine* is responsible for handling queries about both stored context and inferred, higher-

level context.

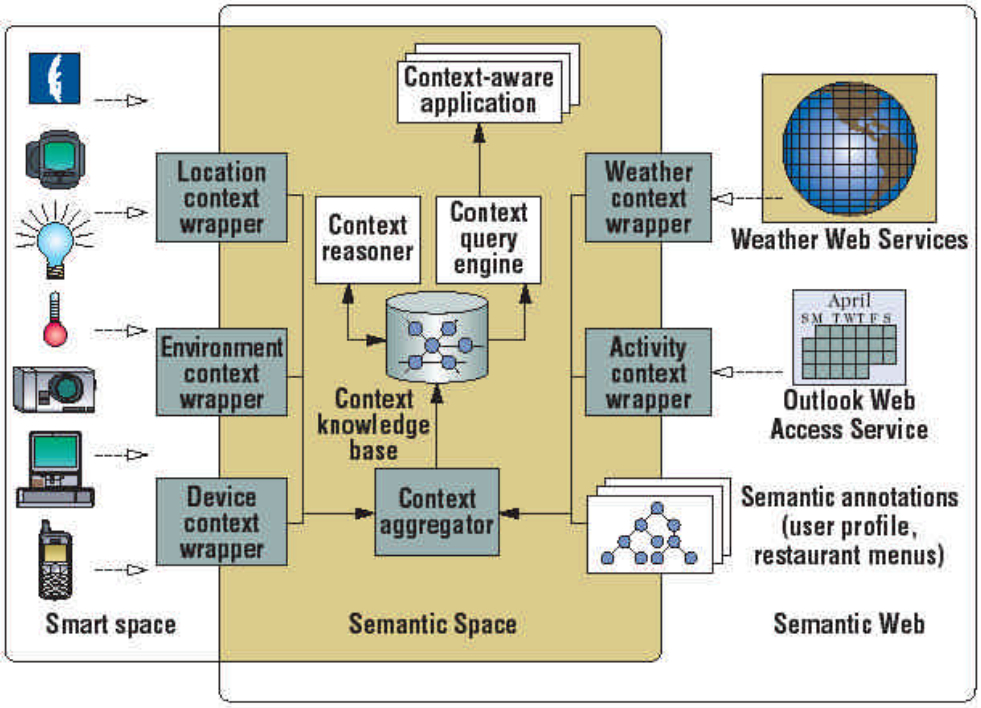

Figure 7: Semantic Space architecture for context-aware applications

4.3 CONTEXT WRAPPER

Context wrappers obtain raw context information from various sources such as hardware sensors and software programs and transform them into context markups. Some context wrappers, including the location context wrapper, the environment context wrapper (which gathers environmental information such as temperature, noise, and light from embedded sensors), and the door status context wrapper (which reports the open or closed status of doors in each room), work with the hardware sensors deployed in the smart space. Software-based context wrappers include the activity context wrapper, which extracts schedule information from Microsoft's Outlook 2000; the device context wrapper, which monitors the status of different networked devices (such as voice over IP or mobile phones); the application context wrapper, which monitors the status (idle, busy, closed) of

applications such as JBuilder, Microsoft Word, and RealPlayer from their CPU usage; and the weather context wrapper, which periodically queries a Weather Web Service (www.xmethods.com) to gather local weather information.

All context wrappers are self-contained and self-configured components that support a unified interface for obtaining context from sensors and providing context markups to applications and aggregator. We implemented these wrappers as Universal Plug and Play services that can dynamically join a local network, obtain IP addresses, and multicast their presence for others to discover. Context wrappers can publish context, and applications can register to be notified of context changes detected by the wrapper.

The use of context wrapper provides an abstract programming abstraction for context acquisition - it helps to (i) hide the specifics of input sensors and software from the application developer, (ii) allow changes with minimal impact on applications, and (iii) provide reusable building blocks. Besides the abstract programming model, the role of context wrapper in transforming raw context into semantic markups is very important in that independently developed applications are able to understand and process context based on its semantics.

4.4 CONTEXT AGGREGATOR

Context aggregator discovers context wrappers and gathers context markups from them. The need for aggregation comes in part from the distributed nature of context, as context must often be retrieved from distributed sensors via software components. Rather than have an individual context-aware application access each distributed wrappers in turn, aggregator gather all context markups, making distributed context available within a single point. Hence the use of aggregator provides an additional separation of concerns between context acquisition and actual context use. A part from separation of concerns,

context aggregation is critical for supporting knowledge-based management and processing tasks, such as expressive query and logic inference of context.

We implemented the context aggregator as an *UPnP control point* that inherits the capability to discover wrappers and subscribe to context changes. Once a new wrapper is attached to the smart space, aggregator will discover it and register to published context. Whenever a wrapper detects the change of context, aggregator is notified and then asserts updated context markups into context KB.

4.5 CONTEXT KNOWLEDGE BASE

Residing in each smart space, context KB provides support for scalable storage and knowledge management of context. A context KB stores the extended context ontology for that particular contextual environment and context markups that are either given by users or gathered from distributed wrappers. It dynamically links context ontology and context markups into a single semantic model and provides an interface for the query engine and reasoner to manipulate correlated context.

It is important to note that context knowledge base is different from a relational database that purely supports storage and query. From a knowledge management perspective, "data" of context is low-level facts provided by input sources. When data is connected based on relations, it can answer the "who", "when", "what", "where" questions of context, for example, "where is the user", "who is in the room", "when will the activity begin". However, according to Ackoff's classic definition, "knowledge" is the application of data and information that is able to answer "how-to" questions [36]. A knowledge base allows the inference of implicit information from explicit one, for example, if the context KB (coupled with inference engine) knows "the user is in the meeting location" and "the current time is within the meeting's scheduled interval", it then can deduce that "the user

is at the meeting".

Contexts in ubiquitous computing environment display very high change rates, so the context aggregator must regularly update the context KB with fresh contexts. The scope of contexts that the context KB manages also changes depending on the availability of wrappers. Application developers can add a new wrapper to expand the scope of context in a contextual environment or remove an existing wrapper when the contexts it provides are no longer needed. The aggregator monitors the wrappers' availability and manages the scope of contexts in the context KB. When a context wrapper joins the smart space, the context aggregator adds the provided context to the context KB, and when the wrapper leaves, the aggregator deletes the contexts it supplied to avoid stale information.

4.6 CONTEXT REASONER

The context reasoner is responsible for inferring higher-level contexts from basic sensed contexts stored in the context KB. Because Semantic Space explicitly represents context, existing general-purpose reasoning engines can directly process this information, making it easy for developers to realize application-specific inferences simply by defining heuristic rules. The use of context reasoner helps to separate the implementation of context inference from individual applications, thus frees application developers from writing code to perform reasoning.

A most important feature of ubiquitous computing applications is customizability. In the same smart space, different applications may define dissimilar (sometimes conflicting) rules for inferring a given type of higher-level context. Since application-specific rules may generate conflicting results, the context reasoner doesn't assert inferred contexts into the context KB, thus avoiding conflict in the coherent model. When an application needs certain higher-level context, it submits a set of rules to the context reasoner, which applies

them to infer higher-level context on the application's behalf, then keeps the newly inferred context in a temporary model without storing them in the context KB. The temporary model then can be accessed by the context query engine to provide higher-level context to the requesting application.

Our current system applies Jena2 to support forward-chaining reasoning over the OWL-represented context. To perform context inference, an application developer needs to provide horn-logic rules for a particular application based on its needs. The context reasoner is responsible for interpreting rules, connecting to context KB, evaluating rules against stored context, and providing interface for the query engine to access inferred result. For example, developers can define application-specific rules to infer a user's likely situation based on context about user, activity and location. The following rule examines whether a given person is currently engaged in a meeting on the basis of location and schedule—if he's in the meeting location and the current time (returned by `currentDateTime()`) is within the meeting's scheduled interval, he's likely to be at the meeting.

```
type(?user,User) ∧ type(?event,Meeting) ∧ location(?event,?room) ∧
locatedIn(?user,?room) ∧ startDateTime(?event,?t1) ∧
endDateTime(?event,?t2) ∧ lessThan(?t1, currentDateTime()) ∧
greaterThan(?t2, currentDateTime())=> situation(?user,AtMeeting)
```

4.7 CONTEXT QUERY ENGINE

Context query engine is responsible for handling expressive query from applications and updating applications with up-to-date query result on a continuous basis. Unlike request-response paradigm where an application poses a query and a query engine processes generate a finite result set, context query engine allows applications to register logical specifications of interest over changing context and receive streaming results

asynchronously.

Furthermore, an application may seek higher-level context (e.g. user situation, room activity) that is not directly available in context KB. In this case, context query engine interfaces with context reasoner to derive higher-level context. For instance, an application seeking user's situation context needs to provide context query engine with two parameters – the query statement specifying application's contextual interests and the logic rules defining the desired way to derive user situation. Upon requested, context query engine will add the application's request in its registration table. To answer the query, the engine first triggers context reasoner to generate the inferred result, from which it then extracts user situation by query processing.

In Semantic Space, we name the above-described support as *Inference-enabled continuous query*, which provides a federated support for expressive query, continuous delivery and context inference.

4.7.1 Query Specification

Semantic Space defines a simple specification language for developers to create inference-enhanced continuous query. The specification has the following form.

```
{CREATE query}
[TRIGGER [rule 1][rule 2]...[rule n]]
[START start]
[EVERY interval]
[EXPIRE expiration]
```

Application developers can define such queries by combining an ordinary query with the inference rule set and additional temporal annotations. The query will become effective at the time given by `start`. The parameter `interval` indicates how often the query is to be executed. If the `interval` is not zero, the inference and query will be triggered whenever context KB is updated. Queries will be deleted from the context query engine

automatically after their expiration time indicated by `expiration`. The query specification can be associated with logic rules (given by **`TRIGGER`**) for inferring higher-level context based on application's need. For example, the example in Figure 8 specifies the query for `XiaohangWang`'s situation, which will be automatically executed every 60 seconds during `20040809T090000` and `20040809T190000`.

```
CREATE
      SELECT ?situation
      WHERE (<ss:XiaohangWang>, <ss:hasSituation>, ?situation),
      USING ss FOR <http://www.i2r.org.sg/semanticspace#>
TRIGGER
      [type(?user,User) ^ type(?event,Meeting) ^ location(?event,?room)
      ^ locatedIn(?user,?room) ^ startDateTime(?event,?t1) ^
      endDateTime(?event,?t2) ^ lessThan(?t1,currentDateTime()) ^
      greaterThan(?t2,currentDateTime())=>situation(?user,AtMeeting)]
      /* More rules are not shown in this example*/
START 20040809T090000
EVERY 60
EXPIRE 20040809T190000
```

Figure 8: Specification for an example *inference-enabled continuous query*

4.8 SUMMARY

This chapter established the architectural requirements for supporting context-aware applications. Following on from these requirements, the chapter then introduced Semantic Space, the generic context-aware architecture that satisfies these identified requirements. In essence, the architecture focuses on providing a level of abstraction between sensors, context infrastructure, and context-aware applications. The design included a description of the constituent components which can be used together to support context-aware applications designed for ubiquitous computing environments. The following chapter details the implementation of the Semantic Space architecture and, in addition, describes how to integrate sensor systems and build context-aware applications using this architecture.

CHAPTER 5
IMPLEMENTATION OF SEMANTIC SPACE

The previous chapter demonstrated the need for an architectural support for context-aware applications and introduced the key components of Semantic Space. The prototype Semantic Space introduced demonstrates a more flexible approach to context-aware application design and systematically supports the features described in section 4.1. Semantic Space is designed to provide the application developer with a convenient mechanism for supporting the entire processing flow, from context capture, aggregation, inference, query, to delivery.

This chapter considers the Semantic Space from an engineering perspective and describes the prototype implementation in detail including application programming interfaces and specifications. The Semantic Space consists of an embodiment of the previously defined architecture and a set of well-defined APIs for supporting the integration of sensors and the development of context-aware applications. Semantic Space architecture was primarily written in Java with roughly 7900 lines of code (except third-party components like Jena2 and JDBC). In this discussion of Semantic Space, we will describe how applications use the architecture, the design of components and how they support the design process and required features.

5.1 COMPONENTS IN SEMANTIC SPACE

Based on the deployment of software components, classes of Semantic Space are grouped into three categories: classes for context wrappers, classes for server architecture, and classes for context-aware applications. Components therefore are grouped into three

packages: (i) `sg.org.i2r.semanticspace.wrapper`, (ii) `sg.org.i2r.semanticspace.server` and (iii) `sg.org.i2r.semanticspace.application`. Among them, package `wrapper` consists of API for implementing software wrapper that automatically integrate context sources into Semantic Space; package `server` contains server-side components that provides essential context functionality (aggregation, storage, inference, query, etc.) together with interfaces used to interact with wrappers and applications; package `application` consists of API that enable distributed applications to access server architecture and wrappers. These components make up the programming abstraction for context-aware application development, allowing developers to think of building context-aware applications in terms of independent logical blocks.

Figure 9 shows how wrappers, applications and the server architecture in the Semantic Space interact with each other. Wrappers are self-contained and self-described components which execute independently without the server architecture, allowing other components discover and subscribe to them using simple interfaces. When wrappers are instantiated, they announce their availability and publish context in form of triples such that other components can select and subscribe to the wrappers with required context. Aggregator in the server architecture discovers all wrappers in the contextual environment asynchronously. To keep context KB consistent with wrappers, aggregator listens to wrapper advertisements and subscribes to new context whenever a wrapper is available. Applications have two basic approaches to retrieve context they need. Applications which make use of context with simple form can directly contact wrapper(s) using subscription-based interface. In this way, applications should specify their contextual interests in form of triple patterns, based on which pattern matching is performed to select the appropriate wrapper. Otherwise, applications with complex context needs, such as the expressive query about correlated context and inference of higher-level context, have to access the functionality provided by the server architecture;

they contact the server, register *inference-enabled continuous query*, and receive updated query result from the server asynchronously.

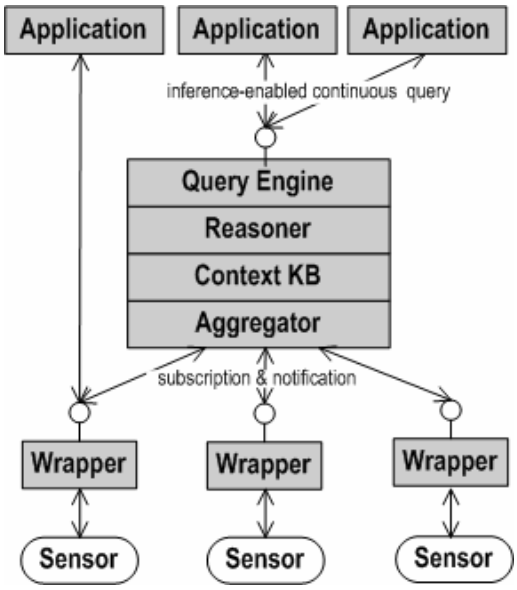

Figure 9: Interaction between wrappers, applications and server architecture

In the remainder of this section, we will discuss these components in detail and describe how they are used by programmers to facilitate context-aware application development.

5.2 WRAPPER

To support the dynamic nature of ubiquitous computing environment, wrapper are designed to aid applications utilize context efficiently whilst offering support for flexibility. In more detail, wrappers acquire captured data from sensor, transform them into semantic markups and publish them for distributed components to access. On the other hand, applications can search for particular wrappers based on their contextual interests, subscribe to the wrappers and get updated asynchronously. To allow for flexibility, application developers may decide at design time which wrappers to utilize without explicitly binding an application to a particular underlying technology. For

example, an application that makes use of location as a form of context could be designed to search the wrapper based on the semantics of context, while the underlying detection and identification technologies (e.g. user login, face recognition, RFID) are transparent to applications. This approach removes coupling between applications and sensors.

5.2.1 Context Triple

To support explicit representation of context, wrappers need to transform sensor data into context markups based on shared ontologies. As described in section 3.4, context markups flowing within the Semantic Space architecture are described in ontology instances, which can be serialized using alternative concrete syntaxes including RDF/XML and RDF/N-Triple. For example, a piece of context markup expressing the weather forecast of a city is serialized into XML (Figure 10 (a)) and triple format (Figure 10 (b)).

```
<City rdf:id="http://...#Singapore">
  <highTemperature>36</highTemperature>
  <lowTemperature>28</lowTemperature>
  <weatherType rdf:resource="http://...#Sunny"/>
</City>
```

(a)

```
(<http://...#Singapore> <http://...#highTemperature> "36")
(<http://...#Singapore> <http://...#lowTemperature>   "28")
(<http://...#Singapore> <http://...#weatherType> <http://...#Sunny>)
```

(b)

Figure 10: XML and triple serialization of the weather context

In Semantic Space, wrappers publish context markups in form of triples, and other components can search for wrappers based on the matching of triple patterns. A triple pattern is a `(subject, predicate, object)` comprised of named variables and RDF values (URIs and literals). To explicitly describe wrapper's capability, each wrapper is associated one or more triple patterns to specify the types of provided context. These triple patterns will be used as service description in wrapper advertisement and discovery. For example, Figure 11 shows that the location wrapper can be described by a single

triple pattern, while the weather wrapper has to be specified by multiple triple patterns. Once a wrapper is started, it periodically sends advertisement message (with triple patterns) on the local network. Due to multicast and the periodic messages, applications are notified about the presence of a wrapper, followed by the process of triple pattern matching and context subscription.

```
(?user, <http://...#locatedInRoom>, ?room)
```
(a)

```
(?city, <http://...#highTemperature>, ?high_temp)
(?city, <http://...#lowTemperature>, ?low_temp)
(?city, <http://...#weatherType> ?weather_type)
```
(b)

Figure 11: Triple patterns for (a) location wrapper and (b) weather wrapper

5.2.2 Developing Wrappers

This section describes the API for wrapper development, and demonstrates the application of this API in efficiently building a sample wrapper for the RFID location tracking system.

5.2.2.1 Wrapper API

The `sg.org.i2r.semanticspace.wrapper` package provides a set of classes and abstract interface (Figure 12) that can be used by application developers to implement a wrapper in a highly-structured, object-oriented way.

Class `Wrapper` represents the key object to construct a wrapper, which is associated with a set of `ContextTriple` objects specifying the provided context and an `UpdateHandler` object implementing actions for context update. Because `Wrapper` is designed to sub-class from standard UPnP service, it automatically inherits the communication and discovery functionalities, being able to support the advertisement and removal of wrappers from the network.

Class `ContextTriple` represents an individual context triple published by the wrapper. While wrappers can provide multiple types of context, subscribing component could be only interested in a subset of them. For example, a wrapper may return three types of context triples, but the subscriber may only be interested in two. Therefore context subscription is based on context triples. The specification of `ContextTriple` requires compulsory parameters (`triple_pattern`) and two optional parameters (`subject_type`, `object_type`). Given these parameters, context discovering components are able to search context triples based on their context needs. This class provides methods to add and remove subscription, as well as to update subscribers when context changes.

The developer needs to implement the `UpdateHandler` interface in order to communicate with sensors and transform acquired data into context triples. To handle simultaneous updates from sensors, `UpdateHandler` uses a multithreaded server. When it receives sensor update event, the server's current thread handles the new request and it forks a new thread to listen for future event. Most commonly, each time the sensor senses new data, the `UpdateHandler` determines whether the new data is significant - this is a wrapper-specific decision. If the data is significant, all subscribers are notified with the updated context triple. Implementation of the `UpdateHandler` interface is dependent on underlying sensing technology applied to obtain data. However, wrapper outputs context in form of triples so as to provide a programming abstraction to hide implementation details from application developers.

```
package sg.org.i2r.semanticspace.wrapper;

/* Define a context wrapper*/
public class Wrapper {

/*Specify advertisement delay. A delay of 0  halts wrapper
 *advertisement, default value is 60 seconds */
  public void setAdvertiseInterval(int advertisement_interval);

/*Specify a implementation of the update handler */
  public void setUpdateHandler(UpdateHandler handler);

/*Define a context triple */
  public ContextTriple getContextTriple(String triple_pattern,
                                  String subject_type,
                                  String object_type);
/*Publish a context triple */
  public void addContextTriple(ContextTriple triple);

/*Cancel the publishing of a context triple */
  public void removeContextTriple(ContextTriple triple);
}

/* Define a context triple published by wrapper */
public class ContextTriple {

/*Instantiate a context triple */
  public ContextTriple(String tripple_pattern,
                   String subject_type,
                   String object_type);

/*Add a subscription */
  public void addSubscription(String host, int port);

/*Cancel a subscription */
  public void removeSubscription(String host, int port);

/*Update all subscribing components when value of triple changes */
  public void updateSubcribers(String subject_value,
                           String object_value);
}

/* Define a interface for handling context update*/
public interface UpdateHandler {

/* Handler callback method for the update of context triple(s)*/
  public void processContextUpdate();

/* Get the reference of the wrapper*/
  public Wrapper getWrapper();
}
```

Figure 12: Classes and interface for wrapper development

5.2.2.2 Wrapper API in Use

A simple example demonstrating the use of `wrapper` API for building the wrapper for RFID tracking system is shown in Figure 13. In this example, the location wrapper is

used to determine when an individual wearing a RFID tag is present at a room deployed with a RFID sensor. First step to develop a wrapper is to instantiate the `Wrapper` object (`line 03`). The developer then creates the `ContextTriple` object, indicating the triple pattern, subject type and object type to describe the type of provided context (`line 04 - 07`). Although this example only shows a single context triple, there would be multiple context triples associated with a wrapper (e.g. the weather forecast example in Figure 11 is involved in three triples).

The wrapper needs to implement `UpdateHandler` interface using specific code to communicate with the sensor system and to transform sensed data into context triple. In this case, we implement the abstract interface using the class `LocationUpdate-HandlerImpl (line 18)`. The callback method `processUpdate()` listens to location event from the RFID tracking system, parses incoming message to set the context triple, and notifies all components subscribing to the triple(`line 28 - 31`). In this example, a large portion of the code is dedicated to communicating with specific sensor systems, while the developer only needs to write about 30 lines of code to realize wrapper's generic functionality. This demonstrates the leverage provided by Semantic Space's programming abstraction in integrating sensor systems.

```
01. public class LocationWrapper {
     public LocationWrapper() {
       Wrapper wrapper = new Wrapper();
       wrapper.addContextTriple(new ContextTriple(
05.        "(?,<http://www.i2r.org/sg.semanticspace#locatedInRoom>,?)",
           "<http://www.i2r.org.sg/semanticspace#User>",
           "<http://www.i2r.org.sg/semanticspace#Room>"));
        LocationUpdateHandlerImpl handler = new
           LocationUpdateHandlerImpl();
10.    wrapper.setUpdateHandler(handler);
       wrapper.setAdvertiseInterval(60);
       wrapper.start();
     }
     public static void main(String[] args) {
15.    new LocationWrapper();
     }
   }
  public class LocationUpdateHandlerImpl implements UpdateHandler{
   ContextTriple location_triple;
20.String USER;
   String ROOM;
     public void processContextUpdate() {
       location_triple = getWrapper().getContextTriple(
          "(?,<http://www.i2r.org.sg/semanticspace#locatedInRoom>,?)",
25.       "<http://www.i2r.org.sg/semanticspace#User>",
          "<http://www.i2r.org.sg/semanticspace#Room>"));
       while (true) {
       /*Sensor-specific code for
         1. Listen for event from RFID tracking system
30.      2. Parse event message and get USER and ROOM*/
        location_triple.updateSubcribers(USER, ROOM);
       }
     }
   }
```

Figure 13: Example of the location wrapper for RFID tracking system

5.2.3 Accessing Wrappers

This section describes the API for context-aware applications to discover and access distributed wrappers, and demonstrates the use of the API with an example application that accesses the location wrapper described earlier.

5.2.3.1 Wrapper Access API

The wrapper access API is a part of the `sg.org.i2r.semanticspace.wrapper` package, which that can be used by developers to implement a context-aware application that utilizes wrappers. Figure 14 shows the classes (`Discoverer` and `TripleProxy`) and the abstract interface (`UpdateListener`) needed in accessing wrapper functionality.

```
package sg.org.i2r.semanticspace.application

/* Define a context discoverer*/
public class Discoverer {

/* Search a context triple and return a proxy/
  public TripleProxy searchContextTriple(String triple_pattern,
                                         String subject_type,
                                         String object_type);

/* Search all context triples and return a collection of proxies*/
  public Collection searchAllContextTriples();
}

/* Define the proxy object of a context triple*/
public class TripleProxy {

/* Get triple pattern*/
  public String getTriplePattern();

/* Get the type of subject*/
  public String getSubjectType();

/* Get the type of the object*/
  public String getObjectType();

/* Get the value of the subject*/
  public String getSubjectValue();

/* Get the value of the object*/
  public String getOjectValue();

/* Set the update listener to handle context update event*/
  public void setUpdateListener(UpdateListener listener);
}

/* Interface to be implemented to handle context update event*/
public interface UpdateListener {

/* Listener callback for handling context update event*/
  public void contextUpdated();

/* Get the proxy of the updated context triple*/
  public TripleProxy getUpdatedTripleProxy();
}
```

Figure 14: Classes and interface for accessing wrappers

Class `Discoverer` enables applications to discover context wrappers. The class allows applications to not have to know *a priori* the wrapper's network address. It also allows them to more easily adapt to changes in the dynamic environment, as new components may appear and old components may disappear at any time. This class inherits the

functionality of UPnP control point to listen for the advertisement message sent by context wrappers and perform matchmaking on triple patterns to find required context triples. The discovery of context is based on context triples. There are two ways to search for context triple: a specific context triple or all context triples. Typically, an individual application uses the method `searchContextTriple()` to search for specific triples that match its context needs. Similar to the specification of context triples, application's context needs is also specified by triple pattern, type of subject and type of object. Because the server architecture needs to find all context within the contextual environment, this class provides the method `searchAllContextTriples()` for the context aggregator to discover all available context triples.

Class `TripleProxy` is the proxy representation of a context triple on application side. `TripleProxy` is instantiated by the method `Discoverer.searchContextTriple()`, which returns a proxy object for the matched context triple. This class provides methods to inspect meta-information including triple pattern, subject type and object type. Each `TripleProxy` is bound with an `UpdateListener` object.

The developer needs to implement the `UpdateListener` interface for each `TripleProxy`, realizing the functionality of the context-aware application. Most commonly, whenever the wrapper-side `ContextTriple` is updated by sensor data, the correspondent `TripleProxy` will be notified and its `UpdateListener` determines how to act upon this context change. Implementation of the `UpdateListener` interface depends on the application's functionality. However, the structured interface provides a programming abstraction for the application to access context wrappers.

5.2.3.2 Wrapper Access API in Use

An example demonstrating the discovery and subscription process for the location

wrapper described earlier is shown in Figure 15. The location subscriber is a context-aware application that utilizes the location wrapper (described in section 5.2.2.2) to acquire user's room-level location. Most commonly the wrapper access process consists of following steps: (1) the wrapper periodically multicasts its advertisement message which contains meta-information about the context triple(s) it publishes; (2) the application receives wrapper's advertisement message and subscribes to the context triple(s) that match its contextual interests; (3) the underlying sensor system detects the change of context (in this case, the presence of a user in a room); (4) the wrapper filters sensor data, updates corresponding context triple based and notifies subscribing applications. Actual code of the example is given in Figure 16.

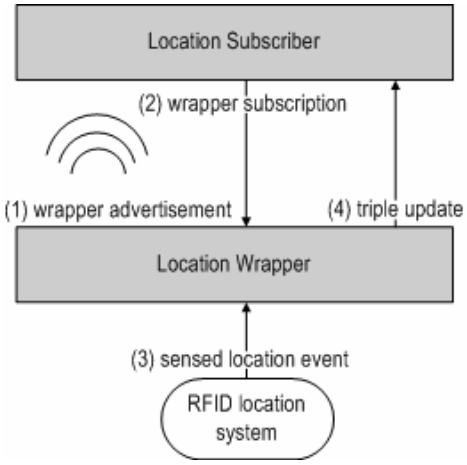

Figure 15: Example of an application accessing a wrapper

In application development, `LocationSubscriber` first instantiates the `Discoverer` (`line 03`), which is then used to locate the context triples that are of interest to them. In this case, it specifies the application's context needs by providing triple pattern (`(?,<http://www.i2r.org.sg/semanticspace#locatedInRoom>,?)`), type of subject (`<www.i2r.org.sg/semanticspace#User>`) and type of object (`<www.i2r.org.sg/semanticspace#Room>`). The `Discoverer` uses this information to search for context triple and returns a proxy of the matched triple (`line 04`). The application then binds the

triple proxy with an `UpdateListener` interface to handle incoming context update (`line 09`).

To realize context-aware behavior of the application, the developer needs to implement `UpdateHandler` interface to decide the actions upon context. Here the class `LocationUpdateListenerImpl` is created (`line 19`) with the callback method `contextUpdated()` listens to context update event sent by the wrapper and perform application-specific action on this information (`line 19 - 27`).

Although this example only shows the application discovering and subscribing to a single context triple, there would be applications that need to utilize several triples published by different wrappers within the contextual environment. The code to implement the access of multiple triples would be similar except that there would be multiple `Discoverer.searchContextTriple()` method calls and `UpdateListener.contextUpdated()` callback needs to handle notification from multiple context triples.

```
01.public class LocationSubcriber{
    public LocationSubscriber() {
      Discoverer discoverer = new Discoverer();
      TripleProxy location_triple_proxy =
05.      discoverer.searchContextTriple(
         "(?,<http://www.i2r.org.sg/semanticspace#locatedInRoom>,?)",
         "<www.i2r.org.sg/semanticspace#User>",
         "<www.i2r.org.sg/semanticspace#Room>");
     location_triple_proxy.setUpdateListener(new
10.      LocationUpdateListenerImp());
    }
    public static void main(String[] args) {
      new LocationSubscriber();
     }
15.}
  public class LocationUpdateListenerImp implements UpdateListener {
    String USER;
    String ROOM;
    public void contextUpdated() {
20.    if (getUpdatedTripleProxy().getTriplePattern().equals
  ("(?,<http://www.i2r.org.sg/semanticspace#locatedInRoom>,?)")){
       USER = getUpdatedTripleProxy().getSubjectValue();
       ROOM = getUpdatedTripleProxy().getObjectValue();
      /*application-specific code to implement context-aware
25.      behavior. The example simply displays received context*/
       System.out.println(USER + "is in " + ROOM);
     }
    }
```

Figure 16: Application code demonstrating the access of location wrapper

5.3 SERVER

Apart from the wrapper abstraction we described in section 5.2, Semantic Space also has a server architecture and an application programming interface for accessing server functionality. As described in CHAPTER 4, the server architecture support richer functionality, including context aggregation, persistent storage, context inference, expressive query and continuous delivery, to complement individual wrappers. It consists of four collaborating components. Context aggregator is responsible for discovering and subscribing to all context triples asynchronously, and keeps updating the context KB when it receives context update. Context KB dynamically links the context ontology and gathered context triples into a single semantic model to support expressive query and logic inference of context. Context KB stores context persistently in the relational database. Context reasoner supports rule-based inference to generate higher-level context from basic context stored in KB. Context query engine provides the support for *inference-enabled continuous query* as described in section 4.7.1. It handles expressive queries to extract context from KB, keeps notifying applications whenever query results change, and triggers context reasoner to execute inference if the application seeks higher-level context.

Semantic Space logically encapsulates underlying processes into a single architecture and provides applications with a simple interface to access server functionality. This approach offloads the burden of complex management and processing tasks from individual applications, thereby offering a programming abstraction for application development.

5.3.1 Implementation of *Inference-Enabled Continuous Query*

The interface between applications and server is query. To give the details of the support

for inference-enabled continuous query, we describe the typical process for an application asking for higher-level context from the server (Figure 17). The example application (`SituationQueryer`) needs to be notified when the situation of `XiaohangWang` changes. It expresses this context needs in form of the query specification (Figure 8, section 4.7.1), providing the RDQL statement and a set of inference rules. One of the rules examines whether a people is engaged in a meeting on the basis of location and meeting schedule - if he's in the meeting location and the current time is within the meeting's scheduled interval, he's likely to be at the meeting.

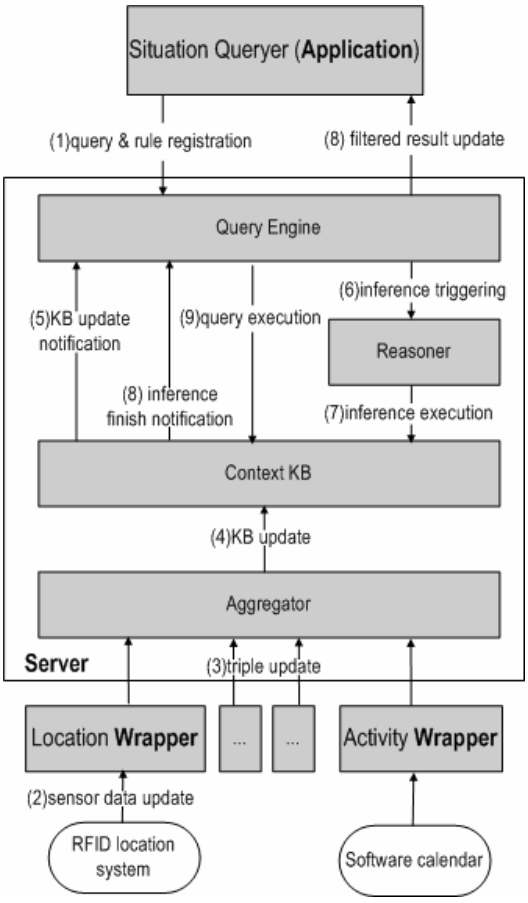

Figure 17: Process for inference-enabled continuous query

The following lists the interaction between the example application and different components. (1) To utilize server functionality, the application first registers the query

specification with the server architecture. (2) When a relevant wrapper (e.g. the location wrapper) has new incoming data from the sensor system, for example, "`XiaohangWang` is present at the meeting location `Room233`", it updates the context triple that represents this sensor data. (3) The wrapper then notifies all subscribing components (context aggregator and individual applications) with the updated context triple. (4) When context aggregator receives the location change, it submits this information to context KB for update. (5) Once context KB finishes the location update in its persistent storage, it notifies context query engine with a KB update event. (6) Because the application needs higher-level context, query engine calls context reasoner to apply context inference. (7) Context reasoner evaluates the application-supplied rule set to generate an inferred result, and keeps it in a temporary KB model. In this case, the situation of user `XiaohangWang` is determined to be `AtMeeting`. (8) Context query engine will be notified when the inference finishes. (9) It then extracts the desired context (`XiaohangWang`'s situation) by querying the temporary KB model. It is also responsible for destructing the temporary KB model after the query finishes. (10) Finally, context query engine compares the new result with the old one, and only sends the changed query result the application.

5.3.2 Accessing Server Functionality

Semantic Space has a simple application programming interface for distributed applications to access server functionality. Through the use of this API, a level of programming abstraction and flexibility is provided to application developers capable of supporting the following features: to facilitate the inference of higher-level context, to selectively access context using expressive query, to support multiple applications accessing the server simultaneously, to reduce the burden of developers to deal with context. This section presents the API and also demonstrates its use with an example context-aware application.

5.3.2.1 Server Access API

The server access API is a part of the `sg.org.i2r.semanticspace.wrapper` package, which is used to implement a context-aware application that utilizes server functionality. Figure 18 shows the included classes and the abstract interface.

```
package sg.org.i2r.semanticspace.application
/*Define an context-aware application*/
public class Application {

/*Instantiate an application, connect to the server*/
  public Application(IP server_host, PORT server_port);

/*Register an Inference-Enabled Continuous Query and return QUERY_ID*/
  public QUERY_ID submitQuery(Query query);

/*Set QueryListener interface to handle the update of query result*/
  public void setQueryListener(QueryListener listener);
}

/*Asynchronous mode query (Inference-Enabled Continuous Query)*/
Public class Query {
  Public Query(String query_statement, /*RDQL query statement*/
     String rule_set, /*Jena2 rule set, null if no inference required*/
     DateTime start_time, /*Start time, null if start immediately*/
     int interval, /*Interval, zero if query is triggered by event*/
     DateTime end_time); /*End time, null if infinite*/

/*Synchronous mode query (query-response mode)*/
  Public Query(String query_statement, /*RDQL query statement*/
           String rule_set);/*Rule set, null if no inference required*/
}

/*Define the object for query result*/
Public class QueryResult {

/*Get the value of the i-th variable*/
  Public Object getVariable(int i);

/*Get QUERY_ID of the query*/
  Public Object getQID();
}

/*Interface to be implemented to handle the update of query result */
public interface QueryListener {

/*Callback method that handles query update*/
  public void resultUpdated();

/*Get the QueryResult that represents the incoming query update*/
  public QueryResult getUpdatedResult();

/*Get a reference of the application bound with this QueryListener*/
  public Application getApp();
}
```

Figure 18: Classes and interface for accessing server functionality

Class `Application` is the representation of a context-aware application. The application uses the method `submitQuery()` to register a query with server. This method returns the `QUERY_ID`, a unique string generated by server to identify the query. The application should be bound with a `QueryListener` to handle the asynchronous notification of result from server.

Class `Query` is used to specify the application's context needs. Semantic Space support two distinct query modes: synchronous mode is used to extract context from server in a query-response manner, and asynchronous mode (or inference-enabled continuous query) is used for server to push streaming results to the application on a continuous basis. Accordingly, there are two constructers: the synchronous query constructor needs the information about query statement and inference rule set (if context inference is involved), the continuous query constructer should be provided with additional temporal annotations. The result of query is represented by class `QueryResult` represents. `QueryResult` records the variables given in **`SELECT`** field of RDQL statement. The method `getVariable(int i)` is used to retrieve the *`i-th`* variable in the result.

The developer needs to implement the `QueryListener` interface for the application, handling the continuous update of each query. Whenever the server detects new result of a registered query, the application will be notified and its `QueryListener` determines how to act upon the change of query result. Implementation of this interface depends on the application's context-aware behavior.

5.3.2.2 Server Access API in Use

To demonstrate use of server access API in application development, we consider the implementation of the `SituationQueryer` example as described in section 5.3.1. The program code of this application is show in Figure 19.

```
01.public class SituationQueryer extends Application {
       public QUERY_ID qid;
       public SituationQueryer (IP server_ip, PORT server_port) {
         Application(server_ip, server_port);
05.      String SituationQuery =
             "SELECT ? situation
             WHERE(<ss:XiaohangWang>,<ss:hasSituation>,?situation)
              USING ss FOR <http://www.i2r.org.sg/semanticspace#>";
         String SituationRuleSet = "[][][]"; //refer to section
10.      Query query = new Query(SituationQuery,
                                 SituationRuleSet,
                                 new DateTime(20040809T090000),
                                 0,
                                 new DateTime(20040809T190000));
15.      qid = submitQuery(query);
         query.setQueryListener(new SituationListenerImp());
       }
       public static void main(String[] args) {
         new SituationQueryer(192.168.137.199, 900 );
20.    }
     }
   public class SituationListenerImp implements QueryListener {
     String SITUATION;
     public void resultUpdated() {
25.    QueryResult result = getUpdatedResult();
       if(result.getQID.equals(getApp().qid)){
      SITUATION = (String) result.get Variable(1);
         /*Application-specific code to implement situation-awareness.
          *This example simply display the update of user situation */
30.     System.out.println("Xiaohang's situation is"+SITUATION);
       }
     }
```

Figure 19: Example application registering the query of user's situation

The application's executable class `SituationQueryer` is extended from the class `Application`, inheriting the generic functionality of query registration and notification. The application first instantiates the `Application` object (`line 04`), and creates a `Query` object to represent its context needs (`line 10`). In this case, the `Query` object specifies a RDQL statement to query `XiaohangWang`'s situation, a rule set to infer user's situation, and temporal annotations to define the activation time, deactivation time and execution interval of the query. Because the application wants to receive new query result whenever it is available, the execution interval is set to zero. The `Query` object is then submitted to server for registration (`line 15`). To handle update events, the developer binds the application with an implementation of the abstract `QueryListener` interface (`line 16`). The implement of the `QueryListener` interface is responsible for realizing the application

context-aware behavior. Here the class `SituationListenerImpl` is created (`line 22`) with the callback method `contextUpdated()` handling the notification of query result asynchronously (`line 24 - 31`).

Although this example only shows the application making use of a single query, there would be applications that need multiple queries for different context. The application code to deal with multiple queries would be similar except that there would be multiple query registrations and the `QueryListener.resultUpdated()` callback needs to handle result update from all queries.

5.2 APPLICATION

A described earlier, the `sg.org.i2r.semanticspace.application` package offers complementary programming abstractions for applications to use context – triple subscription (section 5.2.3) or query registration (section 5.3.2). Typically, the application with simple context needs may discover and subscribe to appropriate triples published by individual wrappers. If the application deals with expressive query or higher-level inference, it can contact the server architecture to utilize inference-enabled continuous query. We have implemented two set of Java APIs (wrapper access API and server access API) to support the two programming abstractions respectively, such that application programmers can prototype a context-aware application in a highly-structured way.

5.3 SUMMARY

This chapter has described the design and implementation of the Semantic Space architecture to provide developers with a useful mechanism for integrating sensor systems and building context-aware applications. The key features of Semantic Space architecture are:

- A set of programming abstractions enabling developers to think of designing sensor components and context-aware applications in terms of logical building blocks.

- A Java API implementation for supporting the rapid prototyping of context wrappers and context-aware applications.

- Utilization of Semantic Web technologies (inference, query, and knowledge base) in realizing the explicit representation, expressive query and logical inference of context.

- The support for *inference-enabled continuous query* that provides a federated solution for query, inference and event notification.

The author believes that these features provide a solution to the problem of supporting context-aware applications designed for use in ubiquitous computing environments. The following chapter evaluates our architecture through the development of a real ubiquitous computing application (*SituAwarePhone*).

CHAPTER 6

CASE STUDY: *SITUAWAREPHONE*

This chapter illustrates how the context model and Semantic Space architecture described in the previous chapters can be used to author a novel prototype context-aware application. The main contributions of this chapter are twofold. First, the general process to build a context-aware application using Semantic Space is introduced and illustrated by examples. Second, the principal research contributions of this thesis (i.e., the aforementioned modeling approach and architectural support for context-aware application development) are evaluated and validated.

6.1 GENERAL DESIGN PROCESS

This section gives the general process that the application developer needs to go through to build a context-aware application.

1. Context Modeling

1.1 Determine the collection of context that is required by the context-aware application to fulfill its functionality.

1.2 If the upper-level context ontology (UCLO) is not sufficient to model all features of the required context, extend UCLO by adding application-specific ontology classes and properties to model these features.

2 Wrapper Development

2.1 If the required context is not available, choose the appropriate physical or software sensor providing it.

2.2 Define triple pattern(s) that acts as the logical representation of the context.

2.3 Follow the design rational described in section 5.2.2 to build the wrapper using Wrapper API, implement the `UdateHandler` interface to acquire data by using the sensor-specific communication interface.

3 Application Development

3.1 Get the required context

3.1.1 If the application wants to make use of individual wrappers to obtain context in simple form

3.1.1.1 Specify the context needs in form of triple pattern(s).

3.1.1.2 Use Wrapper Access API (section 5.2.3) to discover and subscribe to wrapper(s) that match the application's requirement.

3.1.2 If the application has complex context needs and requires expressive query and (or) context inference

3.1.2.1 Specify the context needs in form of RDQL queries.

3.1.2.2 If the application seeks higher-level context that is not directly recognizable from sensors, define rules for inferring this information from sensor-supplied context.

3.1.2.3 Use Server Access API (section 5.3.2) to register inference-enabled continuous query.

3.2 Realize the context-aware behavior by implementing the corresponding interfaces (`UpdateListener` or `QueryListener`) with application-specific code.

To implement the example context-aware application, section 6.4.1 to 6.4.2 will present the respective outcomes of these phases. Before that we introduce the motivations and features of the example.

6.2 INTRODUCTION TO *SITUAWAREPHONE*

The widespread use of mobile phones makes voice communication available anytime, anywhere, but also raises many social problems when, for example, phones ring during meetings or important face-to-face conversations. Normally users often have to configure the settings of mobile phones according to their circumstances to avoid inappropriate usage. However, manual configuration causes frequent interactions with mobile phone, imposing significant user distractions. To advance this matter, we developed a context-aware application, *SituAwarePhone* (Situation-Aware Phone), which automatically adapts mobile phone profiles to the changing situations while minimizing disruption. In this case,

user's situation is used as a form of higher-level context to adapt the behavior of the mobile phone. For example, if a user is determined to be in a meeting then a silent mode may be automatically selected which re-directs all incoming calls to voice mail. However, a user walking in an outdoor environment will have the mobile phone configured so that the volume is set to maximum and the vibrating alert turned on.

To access the situation context, *SituAwarePhone* registers query with Semantic Space architecture, which in turn notifies the phone with the changes of user's situation on a continuous basis. *SituAwarePhone* also may query other types of context that helps in adaptation. For example, it queries the end time of the meeting the user is engaged in to schedule a callback to the caller, or it takes account of the caller's identity and identities of other people in a conversation to decide whether to let the call interrupt it.

One of the key requirements of *SituAwarePhone* is the support for user customizability. Customizability is achieved not only by allowing the users to specify how the mobile phone should response to the incoming call in each situation, but also by allowing them to define their own situation inference rule set for the specific contextual environment and application scenarios.

6.3 IMPLEMENTATION

This section presents the outcomes of the design process (section 6.1) in prototyping *SituAwarePhone*. Along with implementation details, we will also discuss the systematic support provided by Semantic Space for context-aware application development.

6.3.1 Context Modeling

Because *SituAwarePhone* prototype is initially designed for use in I^2R workplace, we need to add additional classes and properties to ULCO in order to model this particular contextual environment. As shown in the top right of Figure 3, the extended context ontology includes the following features. The class `Room` is classified into detailed types including `LabRoom`, `MeetingRoom`, `WashRoom`, `Lounge`, `Pantry` and `PrintingRoom`. To describe typical activities that happen in the I^2R workplace, the abstract class `ScheduledActivity` has concrete sub-classes such as `Meeting`, `WeeklyDiscussion`, `Seminar` and `Interview`. Similarly, the class `AdHocActivity` is sub-classed by `MeetingSupervisor`, `AdHocDiscussion`, `TakingPhoneCall` and so on. To take account of context about devices and applications involved in the application scenarios, we create sub-classes of `ComputingEntity`, such as `MobilePhone`, `FixedPhone`, `MS_PointPoint`, `MS_Word` and `Borland_JBuilder`.

The hierarchical structure of ontologies makes it easy for developers to add application-specific concepts about context. When the application evolves and needs more context types, we can fulfill the application's modeling requirement by extending the abstract classes in ULCO.

6.3.2 Wrapper Development

The second step in the design process involves in choosing necessary sensors and building software wrappers to provide the collection of context identified in modeling phase. Generally, different kinds of context can be either obtained from physical sensors or inputted by software sensors. The following sub-sections will present how to build wrappers for *SituAwarePhone*.

6.3.2.1 Wrapper for Physical Sensors

There are several types of sensor systems that need to be integrated into the prototype for providing required context, such as location of people, location of mobile phones, status of phones (available/busy/off), status of doors (open/closed), noise level of rooms, and so on. As shown in Figure 20, the deployed sensor systems include X-10 door sensors, noise sensors, Bluetooth mobile phone tracking system, and RFID user tracking system. Accordingly we created context wrappers which transform sensor data into context triples for Semantic Space architecture and context-aware applications to use.

Semantic Space allows developers to use an efficient and highly-structured way to write software wrappers, by leveraging off of the Wrapper API. For example, the location wrapper described in 5.2.2.2 only required about 30 lines of code (out of a total of 1600 lines) to deal with generic functionalities, such as wrapper advertisement, subscription, and notification. The standard API also makes it very easy to change the underlying technology used in context sensing. We were able to swap the implementation of the location wrapper entirely when we went from using simulated software sensor (Figure 21 in section 6.3.2.2) to using RFID technology and vice-versa, without changing a line of code in the context-aware application. This ability allows us to easily evolve our systems, in terms of the sensing technologies, and to prototype with a variety of sensors.

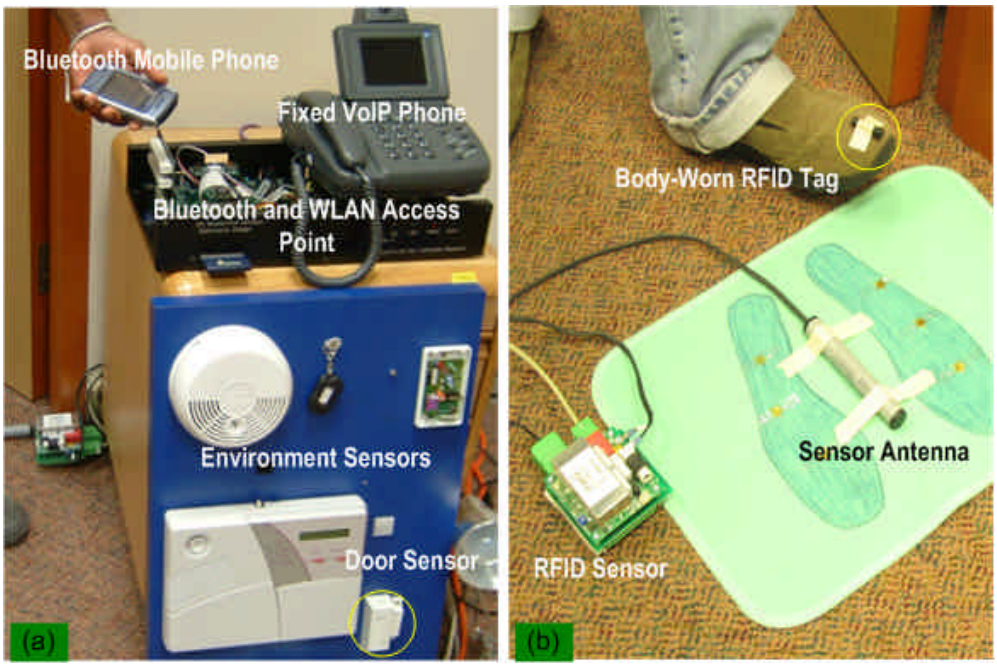

Figure 20: Physical deployment of *SituAwarePhone*: (a) networked sensors and devices; (b) RFID user tracking system

6.3.2.2 Wrappers for Software Sensors

A non-functional requirement of Semantic Space is to support the early prototyping of context-aware applications, even in the absence of physical sensors. Prototyping a new application is often barriered by sensors that are not available for use. In normal cases, if the required sensors are not available, we either have to drop the application idea or determine how to build it without having the necessary sensors. To this end, Semantic Space allows the use of software sensors that collect user input from GUIs to simulate physical sensors. This approach makes it convenient for developers to prototype novel applications in its early stage.

Figure 21 shows a GUI that simulates the RFID user tracking system described earlier. The GUI displays a number of rooms and allows the user to choose one in order to indicate his current location. The simulated GUI input is transformed into context triples (shown in the bottom of the GUI) that can be processed by the architecture and

applications. Because the wrappers for software sensor have the same interfaces as the wrappers for physical sensors, an application can use either interchangeably, regardless of the underlying sensor used, without any change to application code. We also developed other software sensors that are useful to *SituAwarePhone*. Figure 22 shows a GUI that is used to detect the presence of objects (users and devices), noise level and door status for a room, as well as to simulate the status changes of a device. Figure 23 shows a GUI used by users to edit calendar event for providing activity-related context. By treating the GUIs as sensors and wrapping them with the standard interface, the application can be prototyped and tested. When suitable physical sensors become available, they can be deployed without requiring any change to the application.

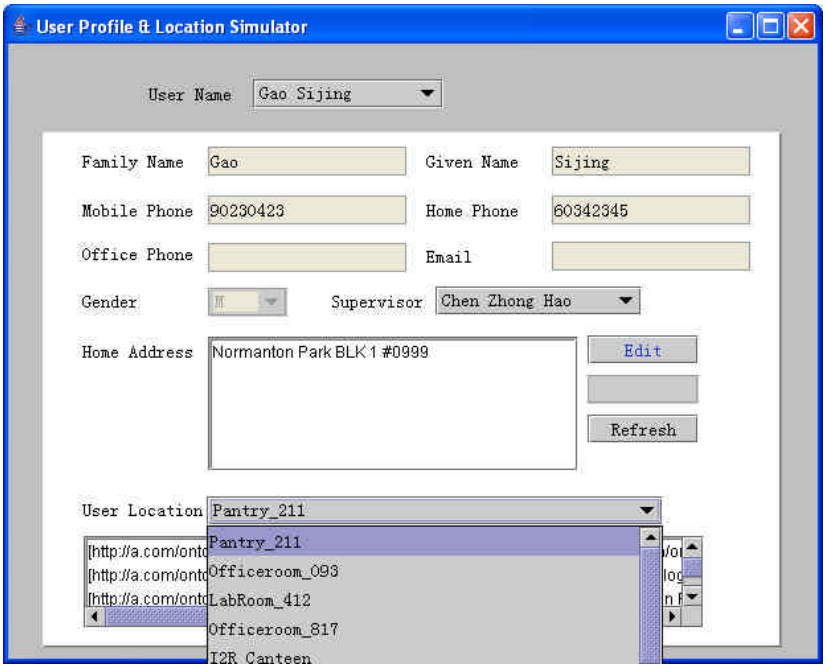

Figure 21: GUI for providing user profile and simulate the location change of people

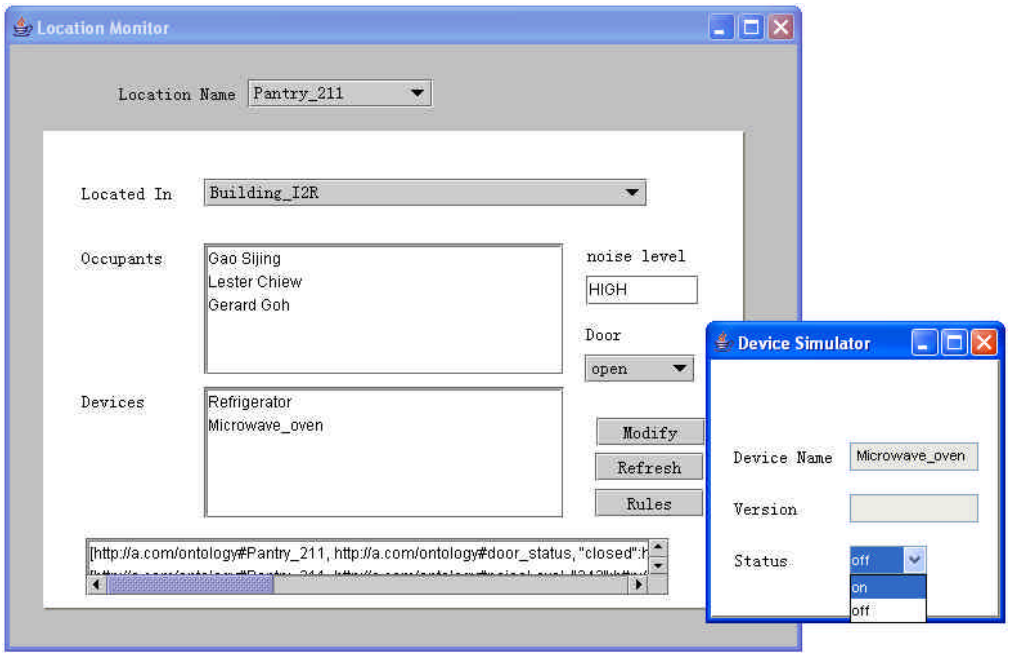

Figure 22: GUIs for simulating the detection of moving objects, noise level, door status of a room, as well as the status of a device

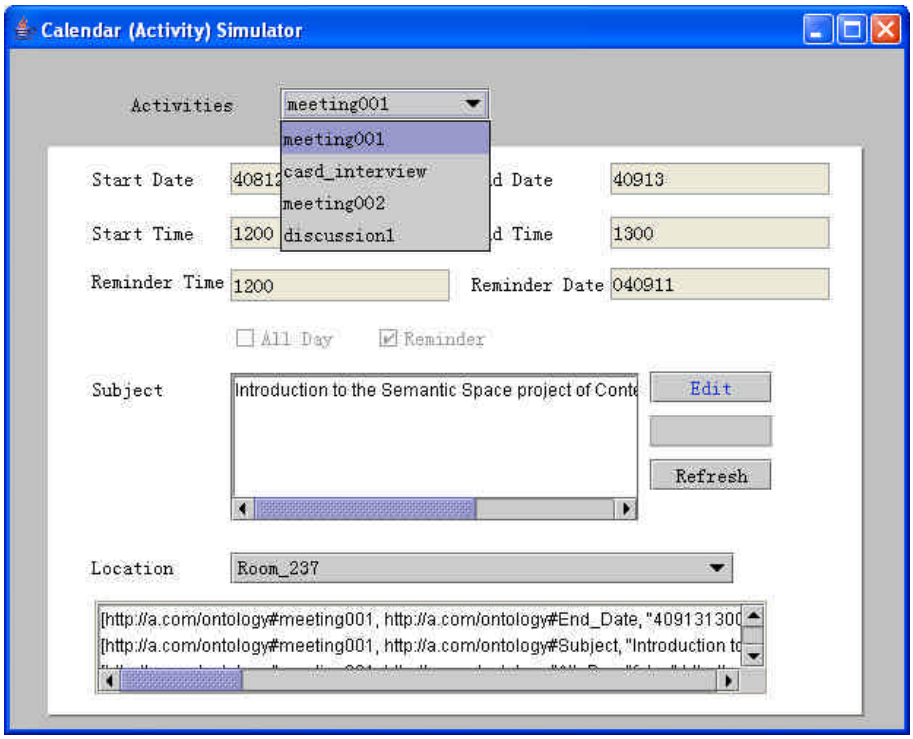

Figure 23: GUI for supplying activity-related context

6.3.3 Application Development

The final step of the design process is application development. Because *SituAwarePhone* involves in query and inference, we design it to get context from the server architecture. Therefore, the developer's efforts in this phase can be divided into two parts: the specification of context needs (in form of *inference-enabled continuous query*), and the implementation of context-aware behavior.

6.3.3.1 Specifying Context Needs

The most important context used by *SituAwarePhone* is the situation of a user, which is a higher-level context that has to be inferred from other basic context obtained from sensors. In order to be notified with the changing situations, the application needs to register an *inference-enabled continuous query* with context query engine, specifying the query statement, situation inference rule set, and temporal information. Figure 24 (a) shows a query specification with a reference rule set to infer a user's situation from basic context about `User` (such as type, identity and relationships with others), `Activity` (such as type and time interval), `Location` (such as meeting location, office location, and other people at the same location), and `ComputingEntity` (such as status and ownership). In using *SituAwarePhone*, users can either choose to apply the reference rule set, or customize the rule set based on their own needs.

Besides continuous query, *SituAwarePhone* also uses simple (synchronous) queries to get other context that is useful in profile adaptation. For example, Figure 24 (b) shows the query used to get the end time of the meeting the user is currently engaged in to schedule a callback to the caller.

```
CREATE
   SELECT ?situation
   WHERE (<ss:XiaohangWang>, <ss:hasSituation>, ?situation),
   USING ss FOR <http://www.i2r.org.sg/semanticspace#>
TRIGGER
   [ AtMeeting:
   type(?user,User ^ type(?event,Meeting) ^ type(?user, user) ^
   location(?event,?room) ^ locatedIn(?user, ?room) ^
   startDateTime(?event,?t1) ^ endDateTime(?event, ?t2) ^
   lessThan(?t1,currentDateTime()) ^ reaterThan(?t2,currentDateTime())
          =>situation(?user, AtMeeting)
   ][ TakingPhoneCall:
   type(?user,User) ^ type(?phone,MobilePhone) ^
   owner(?phone, ?user) ^ status(?phone, Busy)
          =>situation(?user, TakingPhoneCall)
   ][ AtWriting:
   type(?user,User) ^type(?app,MS_Word) ^registeredUser(?app, ?user)
   ^ status(?app, ?busy)
          =>situation(?user, AtWriting)
   ][ AtProgramming:
   type(?user, User) ^ type(?app, Borland_JBuilder) ^
   registeredUser(?app, ?user) ^ status(?app,?busy)
          =>situation(?user, AtProgramming)
   ][ AtLunch:
   type(?user,User) ^ locatedIn(?user,I2R_Canteen) ^
   greaterThan(currentTime(),12:00:00) ^
   lessThan(currentTime(), 13:30:00)
          =>situation(?user, AtLunch)
   ][ UsingWashroom:
   type(?user,User) ^ type(?room,Washroom) ^ locatedIn(?user, Washroom)
          =>situation(?user, UsingWashroom)
   ][ MeetingSupervisor:
   type(?user,User) ^ supervisorOf(?user2,?user) ^ office(?user2,?room)
   ^ locatedIn(?user,?room) ^ locatedIn(?user2,?room) ^
   doorStatus(?room,closed) ^ noiseLevel(?room,?x) ^ greaterThan(?x, 60)
          =>situation(?user, MeetingSupervisor)
   ]
START NOW
EVERY 0 /*Executed immediately when context KB is updated*/
EXPIRE NEVER
```

(a)

```
SELECT ?endTime
WHERE  (?event, <rdf:type>, <ss:Meeting>),
       (?event, <ss:hasLocation>, ?room),
       (<ss:Xiaohang>), <ss:locatedIn>, ?room),
       (?event, <ss:start>, ?startTime),
       (?event, <ss:end>, ?endTime)
AND    (startTime < currentDateTime() && endTime > currentDaytime())
USING  ss FOR <http://www.i2r.a-star.org.sg/semanticspace#>,
       rdf FOR <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

(b)

Figure 24: Query specifications of *SituAwarePhone*: (a) *inference-enabled continuous query* for `XiaohangWang`'s situation; (b) simple query for the meeting's end time

6.3.3.2 Implementing Situation-Aware Behavior

Upon the reception of situation context, certain functionality should be provided to determine the situation-aware behavior to perform as well as indicate how the behavior should be executed. This last step is the main emphasis in building *SituAwarePhone*. Figure 25 shows the snapshots of the prototype implementation.

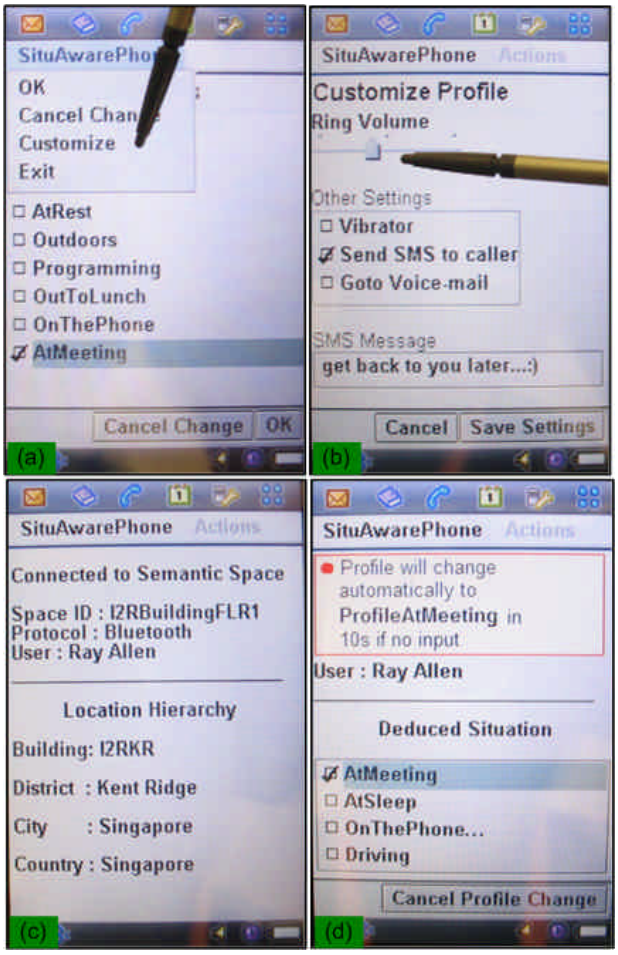

Figure 25: Snapshots of *SituAwarePhone* running on Sony-Ericsson P900 mobile phone: (a, b) profile customization for each situation; (c) connection establishment with Semantic Space architecture; (d) automatic profile adaptation in situation change

SituAwarePhone supports the customization of situation-aware behaviors through the use of simple GUIs, as shown in Figure 25(a, b). Based on J2ME API, we have implemented a set of mobile phone functions (e.g. silence, vibration, volume control, call forwarding

and SMS notification), from which users can choose an appropriate combination for use in different situations. Another consideration for the situation-aware behavior is the balance between the autonomy of the application and sufficient degree of user control. It is commonly agreed that human users should be retained in the decision loop of context-aware applications, while absolute autonomy is undesirable in most cases [37]. As shown in Figure 25(c), *SituAwarePhone* is designed to notify (using vibration) the user when his situation is determined to be different, presenting a recommended profile for this new situation and allowing them to decide the acceptance or denial of automatic adaptation within a given period.

6.4 SUMMARY

The realization of the *SituAwarePhone* prototype used as a case study demonstrated the practical benefits of Semantic Space detailed in the previous chapters when applied to a sophisticated context-aware application. The primary advantage of Semantic Space is that the programming abstractions and architectural components make the prototyping of context-aware applications easy. More specifically, Semantic Space encapsulates a rich set of functionalizes into the server architecture and provides a uniform interface for gathering context from sensors and disseminating context to applications. The API significantly reduces the workload of context-aware application developers. In the prototyping of *SituAwarePhone*, we wrote n lines of Java code, out of which less than 100 lines are used for handling context. Hence, the developer can concentrate on the details of the application and not worry about the details of acquiring and processing context. Furthermore, the implementation details for specific sensors are hidden with the access to functionality obtained through standard interfaces. Since the provision, management and use of context are controlled through the specification of context ontologies, there is no inherent coupling between any particular application and components (the server

architecture or individual wrappers) offering the context of interest. Therefore, Semantic Space supports context sharing across a variety of independently developed components.

The final chapter presents the conclusions of this thesis before identifying several areas of future work, some of which are currently in progress at Institute for Infocomm Research (I^2R) and build upon the ideas presented in this thesis.

CHAPTER 7
CONCLUSIONS AND FUTURE WORK

This thesis has described a context modeling approach and architecture that support the design, implementation, and execution of context-aware applications. In this final chapter, we summarize our research and briefly describe some areas that merit future research.

7.1 CONCLUSION

This thesis has investigated the use of a generic architecture as a mechanism to support context-aware applications destined for use in ubiquitous environments. More specifically, this thesis has identified and considered a set of requirements for the architectural support for context-aware application development. Based on the identified requirements, an ontology-based context model, a prototype architecture, as well as a set of programming abstractions has been introduced and detailed. The architectural support has been evaluated using the *SituAwarePhone* as a case study, a fully operational context-aware mobile application affording personalized profile configuration and situation-based profile adaptation.

CHAPTER 1 introduced the concept of ubiquitous computing, following which the terms context and context-awareness were presented. We then presented the difficulty for individual applications to make use of context. Therefore, by providing flexible mechanisms to support common functionality and rapid prototyping will eventually lead to a further understanding of this area.

CHAPTER 2 presented a survey of the two main research areas (context-aware computing and Semantic Web) which this thesis combines. Related work ranged from projects that used no supporting abstractions for application development to those that

provided systematic support. We learned merits from this study, but also identified potential improvements. Furthermore, we discussed the advantages of utilizing Semantic Web technologies (ontology, query, inference, knowledge base) in supporting context-aware applications.

CHAPTER 3 introduced our ontology-based approach to context modeling. We presented the design of a two-layer context model, which consists of a standard upper-level context ontology for modeling generic contextual concepts and a set of extended context ontologies for describing application-specific contextual features. This model provides a basis for the explicit representation of context, and makes possible the utilization of inference, query and knowledge base in supporting context processing tasks.

CHAPTER 4 explored the use of a generic architecture, namely Semantic Space, for supporting the development of context-aware applications. Initially, a comprehensive set of requirements was derived by analyzing the key aspects detailed in related work. Following this, we described the high-level design of Semantic Space, consisting of a set of collaborating components for supporting the identified requirements.

CHAPTER 5 described the implementation of Semantic Space architecture designed to provide application programmers with a convenient and flexible mechanism for dealing with context-aware applications targeted at ubiquitous computing environments. Semantic Space not only offers the implementation of architectural components, but also provides a set of programming abstractions in the form of APIs for building sensor wrappers and context-aware applications that get context either from individual wrappers or the server architecture.

In CHAPTER 6, we presented important practical results gained through the use of the context model and Semantic Space architecture in the implementation of a novel context-

aware application, *SituAwarePhone*. We presented the general design rational for prototyping context-aware applications, and then illustrated the process and issues involved in application development by example.

In summary, the contributions of this thesis are:

- A novel ontology-based context modeling approach for supporting the explicit representation of context;
- The identification of architectural requirements to support the building of context-aware applications, resulting in the Semantic Space architecture that provides this support;
- A set of programming abstractions, implemented in the form of APIs, to facilitate the development of sensor wrappers and the design of context-aware applications;
- The implementation of a novel mobile context-aware application, *SituAwarePhone*, to validate and evaluate the architectural support provided by Semantic Space.

7.2 FUTURE WORK

We envision several enhancements to Semantic Space. Currently, the system uses the local-area-network discovery protocol, UPnP, to dynamically locate and access context wrappers. In practical deployment, multiple smart spaces that belong to different users or parties with private context might share the same local network. This presents many opportunities for context misuse from both fraudulent context sources and misbehaving applications. To address privacy concerns, we'll incorporate endpoint authentication into the wrapper discovery process [38]. Each component is associated with a URI and public-key certificate, which can be used to prove its identity to all other components. Enabled

by component authentication, context-aware applications can specify both the context wrappers they trust and those they have access to, thus restricting access of private context to appropriate components.

We also plan to provide support for uncertain context, because dynamic context aren't always precisely obtained from sensors. Using the OWL ontology's probabilistic extension [39] we're working on extending context ontologies to capture uncertainty and give Semantic Space the ability to handle it. Enhanced context ontologies will support probabilistic querying with respect to the quality of context (e.g. granularity, precision, freshness, and level of confidence) and the inference of uncertain context using mechanisms such as probabilistic logic, Bayesian networks, and fuzzy logic.

Another potential improvement is context validation. Because context that flows within Semantic Space is formally represented as ontology instances, we can utilize rule-based reasoning to check the logical correctness of context and determine whether certain properties on context are fulfilled. For example, people are likely to agree on that "the doors and windows must not be locked when heavy smoke is detected in a room with people inside". We therefore can set a property on context (status of door, status of windows, smoke in a room, people in a room) can be formally expressed in term of logical rule(s). If the reasoning (applying property checking rules) over the collected context indicates the violation of this context property, certain actions (e.g. unlocking doors and windows, alerting people) should be automatically carried out to handle this revealed error. Context validation is especially valuable in the contextual environments with mission-critical tasks such as emergency, security, and health-care.

APPENDIX A

JENA RDQL GRAMMAR

```
CompilationUnit     ::=    Query <EOF>
CommaOpt            ::=    ( <COMMA> )?
Query               ::=    SelectClause ( SourceClause )?
                           TriplePatternClause ( ConstraintClause )?
                           ( PrefixesClause )?
SelectClause        ::=    ( <SELECT> Var ( CommaOpt Var )* | <SELECT>
                           <STAR> )
SourceClause        ::=    ( <SOURCE> | <FROM> ) SourceSelector
                           ( CommaOpt SourceSelector )*
SourceSelector      ::=    QName
TriplePattern       ::=    <WHERE> TriplePattern ( CommaOpt
Clause                     TriplePattern )*
ConstraintClause    ::=    <SUCHTHAT> Expression ( ( <COMMA> |
                           <SUCHTHAT> ) Expression )*
TriplePattern       ::=    <LPAREN> VarOrURI CommaOpt VarOrURI CommaOpt
                           VarOrConst <RPAREN>
VarOrURI            ::=    Var
                    |      URI
VarOrConst          ::=    Var
                    |      Const
Var                 ::=    "?" Identifier
PrefixesClause      ::=    <PREFIXES> PrefixDecl ( CommaOpt PrefixDecl )*
PrefixDecl          ::=    Identifier <FOR> <QuotedURI>
Expression          ::=    ConditionalOrExpression
ConditionalOr       ::=    ConditionalAndExpression ( <SC_OR>
Expression                 ConditionalAndExpression )*
ConditionalAnd      ::=    StringEqualityExpression ( <SC_AND>
Expression                 StringEqualityExpression )*
StringEquality      ::=    ArithmeticCondition ( <STR_EQ>
Expression                 ArithmeticCondition | <STR_NE>
                           ArithmeticCondition | <STR_MATCH>
                           PatternLiteral | <STR_NMATCH>
                           PatternLiteral )*
Arithmetic          ::=    EqualityExpression
Condition
Equality            ::=    RelationalExpression ( <EQ>
Expression                 RelationalExpression | <NEQ>
                           RelationalExpression )?
Relational          ::=    AdditiveExpression ( <LT> AdditiveExpression |
Expression                 <GT> AdditiveExpression | <LE>
                           AdditiveExpression | <GE>
                           AdditiveExpression )?
Additive            ::=    MultiplicativeExpression ( <PLUS>
```

```
Expression                  MultiplicativeExpression | <MINUS>
                            MultiplicativeExpression )*
MultiplicativeExpression ::= UnaryExpression ( <STAR> UnaryExpression |
                            <SLASH> UnaryExpression | <REM>
                            UnaryExpression )*
UnaryExpression     ::=     UnaryExpressionNotPlusMinus
                    |       ( <PLUS> UnaryExpression | <MINUS>
                            UnaryExpression )
UnaryExpressionNotPlusMinus ::= ( <TILDE> | <BANG> ) UnaryExpression
                    |       PrimaryExpression
PrimaryExpression   ::=     Var
                    |       Const
                    |       <LPAREN> Expression <RPAREN>
Const               ::=     URI
                    |       NumericLiteral
                    |       TextLiteral
                    |       BooleanLiteral
                    |       NullLiteral
NumericLiteral      ::=     ( <INTEGER_LITERAL> |
                            <FLOATING_POINT_LITERAL> )
TextLiteral         ::=     ( <STRING_LITERAL1> | <STRING_LITERAL2> )
                            ( <AT> Identifier )? ( <DATATYPE> URI )?
PatternLiteral      ::=
BooleanLiteral      ::=     <BOOLEAN_LITERAL>
NullLiteral         ::=     <NULL_LITERAL>
URI                 ::=     <QuotedURI>
                    |       QName
QName               ::=     <NSPrefix> ':' (<LocalPart>)?
                            Unlilke XML Namespaces, the local part is
                            optional
Identifier          ::=     ( <IDENTIFIER> | <SELECT> | <SOURCE> | <FROM>
                            | <WHERE> | <PREFIXES> | <FOR> | <STR_EQ> |
                            <STR_NE> )
```

APPENDIX B

JENA RULE SYNTAX

```
Rule     :=   bare-rule .
         or   [ bare-rule ]
         or   [ ruleName : bare-rule ]

bare-rule:=   term, ... term -> hterm, ... hterm /*forward rule*/
         or  term, ... term <- term, ... term   /*backward rule*/

hterm    :=  term
         or   [ bare-rule ]

term     :=   (node, node, node)        /*triple pattern*/
         or   (node, node, functor)     /*extended triple pattern*/
         or   builtin(node, ... node)
         /*invoke procedural primitive*/

functor  :=   functorName(node, ... node)/*structured literal*/

node     :=   uri-ref              /*e.g. http://foo.com/eg */
         or   prefix:localname     /*e.g. rdf:type */
         or   ?varname             /*variable*/
         or   'a literal'          /*either a string or a number*/
         or   number               /*e.g. 42 or 25.5*/
```

BIBLIOGRAPHY

1. M. Weiser. The Computer of the 21st Century. Scientific American, 94-100, September 1991.

2. WordNet: A Lexical Database for the English Language. URL: http://www.cogsci.princeton.edu/~wn/

3. B. Schilit, M. Theimer. Disseminating Active Map Information to Mobile Hosts. IEEE Network, 8, No. 5, 22-32, 1994.

4. P.J. Brown, J.D. Bovey, X. Chen, Context-Aware Applications: From the Laboratory to the Marketplace. IEEE Personal Communications, 4, 58-64, 1997.

5. N. Ryan, J. Pascoe, D. Morse. Enhanced Reality Fieldwork: the Context-Aware Archaeological Assistant. Computer Applications and Quantitative Methods in Archaeology. 1998.

6. P.J. Brown. The Stick-e Document: A Framework for Creating Context-Aware Applications. In Proceedings of the Electronic Publishing '96, 259-272, 1996.

7. B.N. Schilit, N. Adams, R. Want. Context-Aware Computing Applications, In Proceedings of IEEE Workshop on Mobile Computing Systems and Applications. 1994.

8. A.K. Dey. Understanding and Using Context. Personal and Ubiquitous Computing, Special issue on Situated Interaction and Ubiquitous Computing. 2001.

9. J. Pascoe. The Stick-e Note Architecture: Extending the Interface beyond the User, In Proceedings of the International Conference on Intelligent User Interfaces, 1997.

10. S. Fickas, G. Kortuem, Z. Segall. Software Organization for Dynamic and Adaptable Wearable Systems. In Proceedings of First International Symposium on Wearable Computers (ISWC'97). 1997.

11. A.Dey, G. Abowd. Towards a Better Understanding of Context and Context-Awareness. Workshop on the What, Who, Where, When and How of Context-Awareness at CHI 2000. 2000.

12. A.K. Dey. Providing Architectural Support for Building Context-Aware Applications. Doctoral Dissertation, Georgia Inst. Technology. 2000.

13. N. Davies, K. Mitchell, K. Cheverst, and G.S. Blair. Developing a Context Sensitive Tourist Guide. Technical Report of Computing Department, Lancaster University. 1998.

14. M. Lamming, M. Flynn. Forget-me-not: Intimate Computing in Support of Human Memory. In Proceeding of International Symposium on Next Generation Human Interfaces. 1994.

15. D.L. McGuinness, F. van Harmelen. OWL Web Ontology Language Overview. W3C Recommendation. 2004.

16. A. Harter et al. The Anatomy of a Context-Aware Application. In Proceeding of 5th International Conference of Mobile Computing and Networking (MobiCom 99), 59–68. 1999.

17. A.K. Dey, G.D. Abowd, A. Wood. CyberDesk: A Framework for Providing Self–Integrating Context–Aware Services. Knowledge Based Systems 11(1). 3-13. 1998.

18. B. Brumitt et al. EasyLiving: Technologies for Intelligent Environments. In Proceeding of 2nd International Symposium on Handheld and Ubiquitous Computing (HUC 2000). 12–29. 2000.

19. T. Kindberg et al., People, Places, Things: Web Presence for the Real World. In Proceeding of 3rd IEEE Workshop Mobile Computing Systems and Applications (WMCSA 2000). 2000.

20. J.I. Hong, J.A. Landay. An Infrastructure Approach to Context-Aware Computing. Human-Computer Interaction, vol. 16. 2001. p. 97.

21. G. Chen. Solar: Building a Context Fusion Network for Pervasive Computing. Ph.D. Dissertation, Department of Computer Science, Dartmouth College, August 2004.

22. A. Schmidt. Ubiquitous Computing - Computing in Context. Ph.D. Dissertation, Department of Computer Science, Lancaster University. November, 2002.

23. E. Guttman, C. Perkins, J. Veizades, M. Day., Service Location Protocol. Version 2. RFC 2608, June 1999.

24. Sun Microsystems. Jini Architecture Specification - 1.0. Technical Report of Sun Microsystems, January 1999.

25. Microsoft Corporation. UPnP Device Architecture Specification. Technical Report of Microsoft Corporation. November 1999.

26. Y.Y. Goland, , T. Cai, P. Leach, Y. Gu, S. Albright. Simple Service Discovery Protocol 1.0: Operating without an Arbiter. Internet-draft draftcai-ssdp-v1-03.txt, work in progress, October 1999.

27. T. Berners-Lee et al. The Semantic Web. Scientific America, May 2001.

28. World Wide Web Consortium. RDF and OWL as Semantic Web Recommendations. URL: http://www.w3.org/2004/01/sws-pressrelease. html.en 2004.

29. G. Klyne and J.J. Carroll, eds. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation, 2004.

30. D. Beckett RDF/XML Syntax Specification (Revised). W3C. February 2004.

31. T. Berners-Lee, Primer: Getting into RDF & Semantic Web using N3. W3C. 2000.

32. J. Grant, D. Beckett. RDF Test Cases. W3C. February 2004.

33. J.J. Carroll, et al. Jena: Implementing the Semantic Web Recommendations. In Proceedings of 13th International World Wide Web Conference. New York. 2004.

34. L. Miller, A. Seaborne, A. Reggiori. Three Implementations of SquishQL, a Simple RDF Query Language. In Proceeding of First International Semantic Web Conference, Italy, 2002.

35. D. Zhang, X. Wang, et al., OSGi Based Service Infrastructure for Context Aware Connected Homes. In Proceeding of First International Conference on Smart Homes and Health Telematics (ICOST2003), France, 2003

36. R. L. Ackoff. From Data to Wisdom. Journal of Applies Systems Analysis, p 39. 1989.

37. B.N. Schilit, D.M. Hilbert, J. Trevor. Context-Aware Communication. IEEE Wireless Communication. 9(5)46-54. 2002.

38. S.E. Czerwinski et al. An Architecture for a Secure Service Discovery Service. In Proceeding of 5th Annual ACM/IEEE International Conference of Mobile Computing and Networking (MobiCom 99), 1999.

39. Z. Ding, Y. Peng. A Probabilistic Extension to Ontology Language OWL. In Proceeding of 37th Hawaii International Conference of System Sciences (HICSS 04), 2004.