ABSTRACT

Title of Dissertation:	An Intelligent Broker Architecture for Pervasive Context-Aware Systems
	Harry Lik Chen, Doctor of Philosophy, 2004
Dissertation directed by:	Dr. Timothy W. Finin Professor Department of Computer Science and Electrical Engineering

Context-aware systems exploit the use of situational information, or context, to provide relevant information and services to users. A great challenge remains in defining an architecture that supports context-aware systems. Critical research issues include modeling and reasoning (how to represent contextual information for machine processing and reasoning), knowledge sharing (how to enable agents to acquire consistent knowledge from unreliable sensors and agents), and user privacy protection (how to give users control of their private information that the system acquires).

To address these issues, I developed a new agent architecture called the Context Broker Architecture (CoBrA). It uses the Web Ontology Language OWL to define ontologies for context representation and modeling, defines rule-based logical inference for context reasoning and knowledge maintenance, and provides a policy language for users to control the sharing of their private information. Central to CoBrA is a server agent called context broker. Its role is to maintain a consistent model of context that can be shared by all computing entities in the space and to enforce the user-defined policies for privacy protection.

The major research contributions of this work include a broker-centric architecture for supporting context-aware systems, a standard pervasive computing ontology, a reasoning approach that integrates assumption-based reasoning and argumentation for resolving inconsistent contextual knowledge, and a privacy protection mechanism that exploits information granularity adjustment.

To demonstrate the feasibility of CoBrA, I prototyped a context broker in the FIPA platform using the JADE API library. I showed its use in supporting EasyMeeting, a smart meeting room system that provides context-aware services for assisting speakers and audiences. Other contributions include the CoBrA Demo Toolkit (an open source software package for demonstrating various aspects of CoBrA) and the CoBrA Text Messaging Commands (a text messaging interface for mobile users to interact with a context broker via SMS messages).

The lessons learned from this research are as the follows. (i) CoBrA's broker-centric design can help to reduce the time and effort to rapidly prototype context-aware applications. (ii) Ontologies expressed using the OWL language can provide a uniformed solution for context representation and reasoning, knowledge sharing, and meta-language definitions. (iii) Rule-based logical inference can help to develop flexible context-aware systems by separating high-level context reasoning from low-level system behaviors.

An Intelligent Broker Architecture for Pervasive Context-Aware Systems

by
Harry Lik Chen

Dissertation submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2004

For G.

ACKNOWLEDGEMENT

This dissertation marks the end of my journey as a student at UMBC and the beginning of a new journey as a computer scientist in the industry. My accomplishment over the past several years is indebted to many people. Especially, I want to thank Dr. Tim Finin. Tim is a great advisor and a fantastic friend. It was with him that I first learned how to do research and how to write. His favorite quotes “the best idea is to have many ideas” and “don’t worry be crappy” encouraged to me to think outside the box when I do research. Thank you, Tim, I am very grateful.

I also want to thank my wife Gigi. She is the best cheerleader that I ever had. Whenever I was beaten by the stressful paper deadlines or research obstacles, Gigi was always there to refuel my energy. Being a graduate student is hard. Being a productive graduate student with a balanced life is even harder. Without Gigi, my life would have been dull, and my research would have been less productive and less fruitful. Thank you, Gigi, for loving me and supporting me.

Coming to study in the US was a turning point in my life. I had the opportunity to study abroad because of the tremendous sacrifices that my parents have made. They have done everything possible to ensure that I had a good education. For this and much more, I am forever in their debt. Thank you, Mom and Dad.

I owed much to my committee members. Not only they have helped me to define my research project, but also they have given me advice that is invaluable to the rest of my research career. Thank you, Dr. Anupam Joshi, Dr. Yun Peng, Dr. Marie desJardins, and Dr. Yannis Labrou, I really appreciated what you have done.

I believe we are the sum of our own experiences. I want to acknowledge all my friends who have been part of my life experience – at work and at play. Without them being there to experience everything that we have experienced together, in theory, I could not have been the same person. Especially, I want to acknowledge Dennis Kelly and Matt Kelly, who inspired me to study computer science.

TABLE OF CONTENTS

Chapter I.

INTRODUCTION

“The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it.” – Mark Weiser, 1991.

It was more than a decade ago, when Mark Weiser wrote about his vision of “ubiquitous computing” [108]. He described ubiquitous computing as a new computing paradigm in which computing systems are seamlessly integrated into the lives of everyday users. Weiser believes that computing technology should be made ubiquitous for people. Ubiquitous means that using computing technology should be as natural as using non-computing technology, e.g., writing.

The ubiquitous computing vision has motivated many researchers to study how computing technology can be seamlessly integrated into our everyday life. Among those researchers, many believe that context-awareness is an important aspect of the ubiquitous computing vision [20]. For computing systems to be context-aware means that they are capable of providing users with relevant services and information based on the context of the users (where they are, who they are with, what they are carrying, etc.).

The work described in this dissertation is about developing an architecture to support pervasive context-aware systems in a smart space environment. In this document, I use the term “pervasive computing” as a synonym for the term “ubiquitous computing” since they have been widely referenced in the research literature as interchangeable terms. I use the term “software agent” (or agent for short) to refer to the kind of computing entities that can operate without direct intervention by humans or others [109].

A. Context-Awareness in Smart Spaces

Smart spaces (e.g., intelligent rooms, smart conference rooms, smart vehicles) are instances of pervasive computing systems. These systems typically consist of a community of agents that can coordinate and co-operate with each other to provide services to the human users. Some typical applications of a smart space include automatically capturing free-hand sketches and presenting the meeting synopsis to the absent participants [83], teams of autonomous vehicles conducting space exploration and search-and-rescue missions [72], and teleporting graphical user interfaces from mobile devices to stationary desktop computers [9], and assisting researchers to reschedule meetings and seeking replacement speakers [103].

I believe *intelligence* is a necessary property of the smart spaces. In a smart space, the users will be surrounded by a vast amount of computing services and devices. In order for the users to concentrate on their specific tasks, the smart space should attempt to minimize the amount of manual overhead that is required to configure, control, and manage those services and devices. Hence, I believe intelligent behavior and decision making capabilities are essential in the realization of smart spaces.

Context-awareness is a kind of intelligent computing behavior. For the humans, context-awareness is an essential capability for understanding the implicit information that is associated with the activities that they conduct. For example, context-awareness enables a person to follow an ongoing conversation, and context-awareness can help to guide the appropriate behavior of a student when the student enters a classroom. For the computing systems, however, context-awareness is the capability to provide relevant services and information to the users based on their situational conditions (i.e., contexts) [36].

A consensus definition of context is a collection of information that characterizes a person or a computing entity [36]. Within in the smart space environment, I define the notion of context as the following: by context, I mean an understanding of a location and its environmental attributes (e.g., temperature, noise level, light intensity), and the people, physical objects, and computing entities that it contains. This understanding necessarily extends to modelling the activities and tasks that are taking place in a location as well as the beliefs, desires, commitments, and intentions of the human and the software agents involved.

The following are some typical use case scenarios of context-aware systems in a smart meeting room environment:

- As a speaker enters the room, the room detects her presence and reasons about her intention. Knowing she is the speaker, the room concludes that her intention is to give a presentation. Based on the profile information of this person, the room informs the projector device of the URL from which the presentation slides can be fetched. After the slides have been downloaded, the projector device sets up the presentation.

- Seeing her slides are ready, the speaker signals the room that she is about to begin her presentation. At this time, the room senses the lighting condition is too bright for viewing the projected presentation. Consequently, the room instructs the room lights to be dimmed.
- During the presentation, the room detects the absence of some people who previously expressed interest in the presentation. Knowing this information, the room acquires those people's contact information and informs the meeting minutes agent to send copies of the recorded presentation and the meeting minutes to those people.
- As the presentation comes an end, the room reasons if other services should be provided to the speaker. Knowing the speaker is an invited guest, and who plans to catch a flight later in the evening, the room inquires the speaker if a taxi cab should be reserved for taking her to the airport.
- The speaker leaves the conference room but left behind her PDA in the room. The room detects the PDA's presence. Knowing the device is not co-located with its owner, the room sends a notification to the speaker via text messaging.

B. Issues in Building Context-Aware Systems

There are great challenges in building context-aware systems, e.g. developing accurate sensors to acquire information from the physical environment [89, 107, 65], building software infrastructure and tools to interpret and process the sensed information [1, 8, 94], creating data management systems to manage and store contextual data for later retrieval [93, 99], and developing frameworks to address security and privacy issues associated with the context-aware systems [7, 3, 69].

My work addresses the following key research issues: modeling and reasoning (how to represent contextual information for machine processing and reasoning), knowledge sharing (how to enable agents to acquire consistent knowledge from unreliable sensors and agents), and user privacy protection (how to give users control of their private information that the system acquires).

1. Context Representation

In order for computing systems to process the information acquired from the physical sensors, it must be represented in a form that is suitable for machine processing. In a smart space, there are different varieties of contextual information. For example, it may include time and temporal relations, geo-spatial entities and relations, user profiles, social networks, actions taken by the people and the agents, meeting events, security and privacy policies, the beliefs, desires, and intentions of the people and agents.

In the past [94, 36, 4, 29], contextual information is often represented as data structures or class objects using programming languages (e.g., Java or C++). The key problem with this approach is that the represented information lacks expressiveness and extensibility. Some previous research attempted to address this problem by using a meta-language (i.e., XML) to represent contextual information [17, 50]. However, because XML only provides a syntax level of representation and interoperability, it is unable to provide adequate support for semantic representation and interoperability, which is essential to knowledge sharing and context reasoning [21].

2. Knowledge Sharing

In an open and dynamic environment, no single agent will have complete knowledge about its context. Knowledge sharing is an effective mechanism to help agents to build contextual knowledge. Effective communication is required for knowledge sharing. In order to communicate, independently developed agents must share a common ontology and communication language [41].

In the previous systems [94, 36, 4, 29], because contextual information is typically represented using programming objects and ad-hoc data structures, it cannot be effectively shared among the independently developed agents. However, some approach has suggested the use of a shared database repository to facilitate knowledge sharing [36]. In this approach, the shared database schemas are used to function as a shared ontology between different agents. As the database is updated by different agents when new knowledge is acquired, the database becomes a shared knowledge base for these agents. However, there are several problems with this approach.

First, database schemas have limited expressive power to describe the semantic relations between different data sets [34]. This hinders the expressiveness of context representation. Second, because complex semantic relations are difficult to model using database schemas, it is difficult to define an effective mechanism for detecting and resolving inconsistent contextual knowledge. Third, when a database is shared by multiple agents, all the data that is stored in it is accessible by all agents. Without any built-in functions for controlling the access to this data, it creates concerns for user privacy and information security.

3. Context Reasoning

Context reasoning can play different roles in sensing and knowledge sharing. In sensing, context reasoning is a process that reasons over the sensing data to make interpretations about the context. In knowledge sharing, context reasoning is a process that reasons over different contextual knowledge to detect and resolve inconsistent information.

The design and the implementation of context reasoning can vary depending on the type of contextual knowledge that is involved. In the previous systems [17, 110, 63, 28], system implementations often program the logics of context reasoning directly into the behavior of the systems. Because the implementation for understanding what information is present in the context is tightly coupled with the implementation for guiding the context-aware behavior of the system, the developed applications often have rigid implementations and are difficult to maintain [36]. In order to address this issues, we must take a new approach that allows the logic of context reasoning to be implemented in separately from the behaviors of the systems. Rule-based logical inference is a feasible approach to allow this decoupling [25]. However, adopting a rule-based approach creates a new challenge. Given that context will be represented using ontology languages to enable knowledge sharing. Because these languages have different syntax and semantic representations from the rule languages used to enable context reasoning, a challenge is how to effectively integrate the use of these distinctive languages to support context-aware systems.

Furthermore, when agents are built to share knowledge, they will face the problem of receiving inconsistent information. Because different agents have different beliefs about the context, the information that an agent acquires from the others may be inconsistent with what it currently believes. To address this issue, the context reasoning of an agent must include the capability to detect and resolve inconsistent information.

4. Privacy Protection

The most personal information of a user is often the most useful information to a pervasive context-aware system [69]. This information may consist of the location information of a person, the social behavioral patterns of a person, and the preferences and profiles of a person. Because this kind of information can be misused for wrongful purposes, the users in a smart space will express concerns about their privacy.

Policy is an emerging approach to guide the behavior of an agent in an open and dynamic environment [61]. A policy typically consists of a set of rules for controlling the performance of actions. After a system accepts the policy, it agrees to enforce these rules when it performs actions. In the past, computing systems often uses policy for supporting security access controls. In these systems, policies are typically defined by the system administrators to specify the permissions for different agents to access information.

Policy also can be used for privacy protection [33]. However, there are differences between using policy for security access control and for privacy protection. When using policy for privacy protection, the task of policy definition can no longer rest on the shoulders of a system administrator. This is because different users may have different privacy protection preferences, and it is difficult to define one global policy that would suit the preferences of all users.

While users desire privacy protection, they typically do not intend to hide their contextual information completely from the systems. If they do, it would hinder the systems' ability to provide relevant services and information because of lacking contextual knowledge. A useful feature that a policy language should offer is to allow the users to control the granularity of the information that may be shared by the systems. For example, a user may permit the trusted agents to share the name of the room in which she is present, but for any other agents, only the name of the building is allowed to be shared.

C. Thesis Statement

In a pervasive computing environment, the intelligence of context-aware systems will be limited if the systems are unable to represent and reason about context, and users will abandon the most useful context-aware services if they are unable control the sharing of their private information. By developing a broker-centric agent architecture with expressive ontologies, context reasoning procedures, and a policy-based privacy protection mechanism, we can help computing entities to represent and share context, to detect and resolve inconsistent knowledge, and to protect user privacy.

D. The Context Broker Architecture

A major contribution of my research is a new architecture for supporting pervasive context-aware systems in smart spaces. This architecture called the Context Broker Architecture (CoBrA) is designed to address the four key issues in building context-aware systems: how to representation contextual knowledge, how to enable knowledge sharing, how to enable context reasoning, and how to protect user privacy.

Central to CoBrA is an intelligent agent called *context broker*. In a smart space, a context broker has the following responsibilities: (i) provide a centralized model of context that can be shared by all devices, services, and agents in the space, (ii) acquire contextual information from sources that are unreachable by the resource-limited devices, (iii) reason about contextual information that cannot be directly acquired from the sensors (e.g., intentions, roles, temporal and spatial relations), (iv) detect and resolve inconsistent knowledge that is stored in the shared model of context, and (v) protect user privacy by enforcing policies that the users have defined to control the sharing and the use of their contextual information.

CoBrA differs from other similar architectures [93, 95, 29, 83] in (i) using ontologies expressed in the Web Ontology Language OWL to support context modeling and knowledge sharing, (ii) using logic inferences to detect and resolve inconsistent context knowledge that is acquired from unreliable physical sensors, and (iii) using policies and logical inference to protect the privacy of the users.

The CoBrA design expects the context broker to be deployed in a relatively small scale indoor environment (e.g., offices and living rooms). Typically a single context broker is sufficient to support a smart space. However, being the only service provider in the space, the context broker may become the bottleneck of the system. To overcome this problem, a team of context brokers can be deployed to improve system robustness through redundancy (see Chapter III. for further discussions).

1. A Standard Pervasive Computing Ontology

CoBrA defines a set of ontologies for supporting context modeling and knowledge sharing. These ontologies expressed in the OWL language define the modular component vocabularies to represent intelligent agents with associated beliefs, desires, and intentions, time, space, events, user profiles, social networks, actions, and policies. In the CoBrA prototype implementations, these ontologies were used to support the representation and the reasoning of context in a smart meeting room environment. The details of this ontology is described in Chapter IV..

Key contributions resulted from the ontology development are as the follows. (i) I demonstrated that the use of the ontology languages such as OWL and RDFS, as oppose to the programming languages such as Java and C++, can greatly improve the expressiveness of the representation and the modeling of context. (ii) I showed the OWL language not only can be used to express the semantics of the information on the Web, but also can be used to express the semantics of the contextual information in the physical world.

Because the information in two distinctive domains can be expressed using a common language, the OWL language can help the smart space applications to exploit knowledge in both the Web and the physical world. (iii) I demonstrated the OWL language can be used to define meta-languages (e.g., policies) as the interface languages for the traditional rule-based systems (e.g., policy reasoners). Because RDF/XML is one of the normative surface syntaxes of OWL, and there are great software tools support for editing, managing, and storing OWL data in RDF/XML, by having an OWL interface language for the rule-based systems can help to ease the development efforts and to gain wider acceptance of the systems. (iv) I helped to develop a standard ontology for support pervasive computing applications based on the core ontologies in CoBrA. This ontology is called SOUPA – Standard Ontology for the Ubiquitous and Pervasive Application, and it is currently maintained by the Semantic Web in UbiComp SIG (`http://pervasive.semanticweb.org`).

2. Inference Procedures for Context Reasoning

In CoBrA, there are two types of inference procedures for context reasoning. One for aggregating and interpreting the data acquired from the physical sensors, and the other for detecting and resolving the inconsistent knowledge. I prototyped these inference procedures using different rule-based systems (i.e., Prolog and Jess) and programming API (i.e., Jena), and showed that they are feasible for supporting the context reasoning implementation of CoBrA.

The prototyped reasoning procedures can infer various types of contextual knowledge that are expressed using the SOUPA and the CoBrA ontologies. This includes the temporal relations between different temporal entities (i.e., time instant and time interval), spatial relations between different geographical entities (country, state, city, building, room, etc.), relations associated with people's social networks, the basic states of scheduled meetings and associated properties, and the ownership relations between devices and users and their respective profiles.

To detect and resolve inconsistent knowledge, the context reasoning procedures are implemented with a hybrid design. The OWL-based ontology reasoning is used to detect knowledge inconsistency and the assumption-based reasoning is used to resolve knowledge inconsistency. The OWL-based ontology reasoning exploits the OWL-DL axioms [80] and is built on the existing OWL inference engines (i.e., Jena OWL reasoner [92] and Racer [48]). The inconsistent knowledge is detected by computing RDF statements that are inconsistent within a set of predefined ontologies.

The assumption-based reasoning for resolving inconsistent knowledge is built on the Theorist framework

[87], a Prolog meta-interpreter for default and abductive reasoning. In this approach, context knowledge is treated as an agent's belief about its *observations* of the physical world. Inconsistent context knowledge is formed when the agent has beliefs about two or more conflicting observations. To resolve inconsistent knowledge is to find the *most believable* observation among all the conflicting ones.

In this work, I demonstrated the feasibility of coupling rule-based systems with the semantic web programming framework (e.g., Jena) to support context reasoning. Comparing to the past approaches that use programming class objects to model context and "hardwire" procedures to interpret context, the rule-based approach improves the flexibility of context reasoning implementation. The details of context reasoning's design and implementation are described in Chapter V..

3. A Policy-Based Approach for Privacy Protection

CoBrA uses policy for privacy protection. In this approach, users define policies to control the sharing of their private information and send these policies to the context broker as they enter a smart space. Before the context broker shares a user's information with other agents, it consults the user's policy to check whether or not the communicating agents are permitted to acquire this information. The privacy policies are defined using the SOUPA policy ontology.

The SOUPA policy ontology is designed to be reasoned with a description logic reasoner. The ontology is defined using the DL (description logic) constructers of the OWL language. When defining a policy, the user specifies the actions that represent the sharing of their private information. For each action, the user specifies the prerequisites of the agents that are permitted or forbidden to perform the action. When this policy is reasoned by a description logic reasoner, these ontology descriptions are used to classify the knowledge sharing actions.

The SOUPA policy ontology also allows the users to control the granularity of the information that can be shared with the agents in the space. When some user information is forbidden to be shared and the sharing of a less specific version of the same information is desirable, the context broker will attempt to adjust the granularity of the information. For example, when sharing the specific location of a person is forbidden, the context broker will attempt to adjust the granularity of the location information by finding a more general location concept in a geo-spatial ontology.

The contributions of this work are as follows. (i) I defined a new policy language expressed using the OWL language and showed it can be used to support privacy protection in a pervasive context-aware system.

(ii) I developed a novel algorithm for reasoning over privacy policies and prototyped an implementation of this algorithm using the Jena API and the Racer inference engine. (iii) I developed an ontology-driven approach to facilitate the adjustment of information granularity and showed it can enhance knowledge sharing while avoiding the violation of privacy policies.

E. Minor Contributions

To show the feasibility of CoBrA, I implemented prototype systems to demonstrate various aspects of the architecture. These systems include (i) EasyMeeting – a smart meeting system built on an earlier pervasive computing system developed at UMBC, (ii) CoBrA Demo Toolkit – a collection of agents, user interface programs, ontologies, and policies for demonstrating different aspects of CoBrA, (iii) CoBrA Text Messaging Commands (CTMC) – a text messaging interface for mobile users to interact with a context broker via SMS messages.

1. EasyMeeting: A Smart Meeting Room Prototype

EasyMeeting is an extension to Vigil [106], a third generation pervasive computing infrastructure developed at UMBC. While security is the main focus in Vigil, context-awareness is the main focus in EasyMeeting. In EasyMeeting, the context broker acquires contextual information from the sensors in the room and share that information with various Vigil meeting services. Using this information, the services are able to compose and play personalized greeting messages as people enter the meeting room, help speakers to set up their PowerPoint presentations, and show the audiences the profiles of individual speakers while they are giving presentations.

2. CoBrA Demo Toolkit

CoBrA Demo Toolkit is an open source software package for demonstrating CoBrA. This package consists of a set of agents implemented using the Jade API library. One of these agents is Context Broker, which is capable of acquiring and reasoning with the contextual information. Another key agent is Script Play Agent, which is a configurable user interface program for driving the CoBrA demonstration. I also developed an Eclipse Plug-in module for browsing the ontology and the inferred knowledge that is stored in the context broker's knowledge base. In addition, this toolkit includes a collection of ontologies and privacy policies that are used to facilitate the EasyMeeting and the user privacy protection demonstrations. The CoBrA Demo Toolkit is available at `http://cobra.umbc.edu`.

3. A Text Messaging Interface for Knowledge Query

The CoBrA Text Messaging Commands comprise a list of text messages for mobile users to communicate with a context broker via SMS. The goal is to create a light-weight communication interface for the human users to communicate with the smart space environment. Because the use of text messaging is becoming ubiquitous for the mobile users (e.g., seeking new friends, interacting with the online special interest groups) [84], it is natural to include text messaging as an alternative means to communicate with the smart space services. For example, when users enter the smart space, the system automatically sends the users a list of CTMC commands for controlling various services that are relevant to their situational tasks (e.g. forwarding calls, adjusting the room temperature). I prototyped a CTMC service that allows mobile users to inquire event schedules of the UMBC eBiquity weekly meetings and register themselves to be notified when new presentations and meeting minutes are posted on the eBiquity group web site.

Chapter II.

BACKGROUND

A. Reasons to Study Context

We humans often exploit context when we communicate and take actions. During a conversation, we can often convey ideas to the others without needing to explicitly state the background information. This is because we share a common understanding of the context. For example, when two people walk into a room with only one door, one person tells the other person, “close the door, please”. Because they share the same context, the first person can convey his desire without explicitly pointing to the door that he wants to be closed. In addition, an understanding of context can also affect our behavior. For example, when watching a movie in a movie theater, reasonable people will try to avoid loud conversations; when speeding down on a highway, reasonable drivers will reduce their driving speed if they see a police car is driving next to them.

It is clear that part of the human’s intelligence involves context-awareness [39]. The context-aware subject has been studied in different fields of computer science, which include artificial intelligent [74, 105], information retrieval [19], nomadic computing [63], sensor networks [97, 89] and human-computer interactions [93, 37, 2]. In these research projects, the same subject is studied with distinctive aims. Some projects focused on defining theoretical foundations for context-awareness, and some others are more interested in building applications that exploit context. Nevertheless, these studies all agree that context-awareness should be an essential feature of the future intelligent computing systems.

B. Definition of Context

In order to build context-aware systems, we must define context. What is context? What kind of information is part of or not part of context? How should we describe context from an engineering perspective?

According to the Merriam-Webster Online Dictionary (`http://www.m-w.com/`), context is defined as “(i) the parts of a discourse that surround a word or passage and can throw light on its meaning; (ii) the interrelated condition in which something exists or occurs”. Researchers argue that this definition is inadequate for describing the relation between context and a computing environment [20].

Schilit *et al.* [94] characterize context as a collection of information that describe the users in a context-aware system. Their definition of context is as the follows.

Definition 1 *In a mobile distributed computing system, contexts are the location of the user, the identity of people and physical objects that are nearby the user, and the states of devices that the user interact with.*

While this definition characterizes the types of context that are used in Schilit’s system [94], but it excludes other types of context (e.g. the intentions and desires of the users, the profiles of the users). Dey [36] argues that a definition of context should not just be a list of information that describes users or the system because “*context is all about the whole situation that is relevant to an application and its set of users*”. He gives a different definition:

Definition 2 *Context is any information that can be used to characterize the situation of an entity. An entity is a person, or object that is considered relevant to the interaction between a user and an application, including the user and application themselves.*

Definition 2 is more general than Definition 1 because it is not bounded to a specific list of information. Definition 2 extends the definition to include all information that describes physical objects and computing applications.

I define context as the following:

Definition 3 *Context is information about a location, its environmental attributes (e.g., noise level, light intensity, temperature, and motion) and the people, devices, objects and software agents that it contains. Context may also include system capabilities, services offered and sought, the activities and tasks in which people and computing entities are engaged, and their situational roles, beliefs, and intentions.*

C. Aspects of Context-Aware Computing

1. Enhancing User Interfaces

One aspect of context-aware computing is to enhance the user interfaces of mobile devices. Mobile devices are important part of our everyday life. They give us the freedom to communicate with people and do work in an any-time-anywhere fashion. However, because the size of typical mobile devices shrinks, designing adequate user interfaces for these devices becomes a challenge.

Today's devices often have tiny buttons, small display screens. The users often feel awkward to use these miniature interfaces to interact with the devices. For example, finding a button or activating a control on the screen can require significant visual attention [51].

To address this issue, Hinckley *el at.* have developed a context-aware approach to enhance the user interfaces of a palm-size PC (i.e. Cassiopeia E-105) [51]. In their approach, various functions of a device can be activated based on how a user holds the device. The on-board context-aware system controls the device's functions by calculating the position and the orientation of the device. For example, when the system detects the user is holding the PDA as if he/she is holding a microphone or a cellphone, it automatically actives the voice memo application. When the system detects the device is being held in a landscape orientation, it automatically reformats the display to suit the current viewing orientation.

Independently, Schilit *et al.* [95, 94], Rekimoto [91], Harrison *et al.* [96] have developed similar approaches to enhance the different user interfaces of mobile devices. Schmidt *et al.* describe a cellphone that combines tilt, light, head, and other sensors to detect whether the device is sitting on a table, in a briefcase, or being used in an outdoor environment. Knowing the cellphone is one of those states, the context-aware systems automatically adjusts the ring tone and volume of the cellphone. Rekimoto uses the tilting position context of a device to guide menu selections. Harrison *et al.* exploit the tilting position context to help users to scroll through information that is displayed on the mobile devices.

2. Guiding Systems' Adaptation Behavior

In an open and dynamic environment, the availability of computing resources and conditions may frequently change. Context can help to guide the adaptive behavior of computing systems.

Network properties are commonly used to guide the adaptive behavior of mobile applications. These properties include network bandwidth, error rate, connection setup time, usage costs, security requirements, contention, disconnection rate, and round-trip delay [95]. Among these, network bandwidth is the mostly frequently. For example, while streaming a video file over a wireless network, if an application is able to detect changes in the network bandwidth, it can adjust the streaming quality of the video without interrupting the viewer's attention [95]. Network bandwidth can also help a context-aware web browser to decide the types of image files that it should request from the web server in order to maintain an acceptable downloading speed [76].

3. Enabling Smart Space Applications

Context is also used in building smart space applications. Researchers believe that the use of context can help computing systems to anticipate our needs and act on our behalf. Context-aware systems will draw computing into the natural world of the humans, as oppose to drawing humans into the complex world of computers [49].

Location and user identity are commonly used in smart space applications. In Cyberguide [36], the system provides users with customized direction service and interactive map service by understanding the present location of the users. In a different project [9], a call forwarding system uses a person's location information to decide where the incoming calls to the person should be routed to. Asthana *et al.* [4] describes the use of user identity in a shopping assistant system. In this system, the shoppers identities are tracked by a Person Shopping Server. The server uses this information to provide services such as recommending products and helping the users to locate shelved items.

In addition to location and user identity information, some researchers also explored the use of users' mental states (i.e., intention, desire, and belief) in smart space applications. Unlike location and user identity, the mental states of users are relatively difficult to acquire via sensing. To acquire this information, context-aware systems typically rely on machine reasoning. In the Intelligent Room project [28], the system has a pre-defined behavior model of a user, and this model is used to infer the user's intention. For example, a user's behavior model might be that the user is lying down on a couch, and the user's intention associated with this model might be that the user intends to take a nap. If the system detects the user is currently lying down on a couch, based on the associated intention, the system will try to close the curtains in the room, so that it creates a comfortable resting environment.

D. Context Acquisition Methods

The process to acquire context is called context acquisition [36]. Typically computing systems acquires contextual information via sensors. These sensors may be hardware sensors (e.g. temperature sensors, weight sensors, and noise sensors). Context sensors may also be software programs that aggregate the general information acquired from the hardware sensors to form more specialized knowledge. Different context-aware systems explore different architecture designs and methods to acquire context. I define the following three categories of context acquisition methods:

1. Direct access to hardware sensors

2. Facilitated by a middle-ware infrastructure

3. Acquire context from a context server

1. Direct Access to Hardware Sensors

The rapid advancement in sensing technology is a key driving force behind the context-aware research. In the past, due to technological limitations (e.g., battery life, sensing accuracy), the kind of contextual information that could be acquired via hardware sensors is very limited. Thus, the functions of the context-aware systems that built on these sensing technologies were also limited. Today, as the sensing technology advances, contextual information that previously could not have been acquired now can be directly access by pervasive context-aware systems to enhance their functionalities [40].

Many context-aware systems acquire context via the direct access to physical sensors. A key benefit of this approach is that the high-level applications can have great controls over the operations of the low-level sensors, and can have better knowledge about how different data is collected and computed. In the context-aware Pocket PC designed by Hinckley *et al.* [51], the context-aware system acquires the states and the position of the device by directly accessing the on-board sensors – proximity range sensors, touch sensors, and tilt sensors. From the proximity range sensors, the system acquires a proximate distance between a physical object in the range and the device. From the touch sensors, the system acquires whether or not a user is holding the device and the amount of time that it has been held. From the tilt sensors, the system acquires the tilt angles of the device (i.e. left/right and back/forward), the display orientation of the device, and whether or not the device is being shook.

In the Forget-me-not project [67], the user's handle device accesses the user's location information by communicating with the Active Badge sensors. The same technique is also used in the call forwarding system described by Want *et al.* [107] and the teleporting system described by Bennett *et al.* [9]. In the RFID Chef application described by Langheinrich *el at.*, a cooking recipe recommendation agent is connected to a RFID (Radio Frequency Identification) reader to determine the identity of the cook.

Direct sensor access has some shortcomings. To communicate with different sensors means an application must maintain the implementation to communicate with different sensors. As the number of context used by the application increases, the amount of implementation for sensor communications also increases. For this reason, the application's overall implementation becomes difficult to maintain [36].

2. Facilitated by a Middle-ware Infrastructure

Using context acquisition middle-wares can address the shortcomings of direct sensor access. The idea is that instead of letting the applications to manage the low-level sensing details, middle-ware infrastructures are provided to facilitate sensing. Using middle-ware infrastructures, context-aware applications' implementations can focus on how to use context but not on how to acquire context.

Context acquisition middle-wares are typically built into the hosting devices or platform on which the context-aware applications operate. In the Odyssey project [76], context-aware agents are built on a middle-ware infrastructure to acquire status about the communication network. The middle-ware infrastructure runs on the same hosting device as the agent. In this project, Noble *et al.* successfully demonstrated three different context-aware agents that can adapt their behaviors according to the network bandwidth changes.

Context Toolkit [36] is a middle-ware infrastructure for supporting context acquisition. Unlike the middle-ware infrastructure in Odyssey, Context Toolkit is aimed to provide a general solution for building scalable and reusable context acquisition modules. The Context Toolkit design built on the widget concept in the graphical user interface design. The toolkit defines various kind of widgets (i.e., context widgets) for acquiring context. Widgets shield the low-level sensing implementations from the high-level applications. The user of a widget only concerns the contextual information that the widget provides not the actual operations associated with context sensing.

One problem with the middle-ware context acquisition approach is that it imposes additional computation burden on the hosting devices. A middle-ware design trades computation resources for development convenience. In order to maintain a generic programming interface between the high-level applications and the low-level sensors, certain amount of computation resources (e.g., CPU power, memory, network bandwidths) must be allocated for the middle-ware operations. While this might not create a problem for devices that have rich computing resources, however, it will lead to resource contention problem in devices that have less resources (e.g., cellphones and embedded devices).

3. Acquiring Context from a Context Server

If the hosting device of a context-aware application has limited computing resource, an alternative context acquisition method is to use a context server. A context server is a computing entity that provides contextual information to different context-aware applications in a distributed environment. Instead of building the context acquisition procedures into the devices that host the context-aware applications, the idea of a context server is to shift the context acquisition procedures into the implementation of a server entity that runs on a resource-rich device. From the context server, applications that do not have built-in sensing capability can acquire context and become context-aware.

The Me-Centric Domain Server developed by Perich [81] is an example of a context server. In this system, the physical world is divided into a set of micro-worlds. Each micro-world represents a particular domain in the physical world. A domain might be an office room, a meeting room, etc. Each micro-world has context. The role of a Domain Server is to maintain the context of the individual micro-worlds and share this information with the computing entities of a Me-Centric application.

There are similarities between the design of CoBrA and the Me-Centric Domain Server. In both architecture, a server entity is responsible to share contextual information with other agents in the space. Both the context broker and the Domain Sever use knowledge representation languages to express contextual information (i.e., CoBrA uses OWL, and Me-Centric Domain Server uses RDF).

However, there are differences between these two architectures. In CoBrA, the role of the context broker is not only to share contextual information but also to maintain a consistent model of the context. Maintaining context means to detect and resolve inconsistent information that may have been acquired from unreliable sensors and agents. CoBrA also differs from the Me-Centric Domain Server in the language that it uses to represent context. Using the OWL language to represent context as ontologies, CoBrA allows independently developed agents to share knowledge and provides a means for context reasoning. In addition, CoBrA addresses the privacy issue by providing users with a policy language for controlling the sharing of their contextual information. Privacy was not addressed in the Me-Centric Domain Server.

Chapter III.

ARCHITECTURE DESIGN

A. Characteristics of a Pervasive Context-Aware System

The future pervasive computing environment will be very different from today's computing environment. These differences include a physical environment that is embedded with different types of context sensors, a dynamic network in which devices and services can dynamically join and leave, and an open communication infrastructure that enables independently developed agents to establish relationships and to share information with each other. In order to develop an architecture to support pervasive context-aware systems, we must consider the distinct characteristics of the future computing environment.

Constrained Resources of a Single Agent

In a real-world computing environment, software agents often have a limited amount of resources. In particular, this is true for the agents of pervasive computing. Agents of physical sensors, mobile devices and embedded devices are typically resource poor. While the advancement in computing technology has helped to improve the capability of these agents – e.g., giving them more memory, CPU power, and accurate sensors, it only solves part of the problem.

There is no doubt that advanced technology can improve an agent's capability to acquire and understand context. However, even with the enhanced capability, it is unreasonable to presume any single agent will have sufficient resources to acquire and understand all contextual information that is available in the environment. For example, when an agent must perform multiple context acquisition tasks simultaneously, and a combination of these tasks requires an amount of resources that excesses the agents' capability, the agent may be forced to abandon the performance of certain tasks. Consequently, the agent is unable to develop a complete understanding of the context.

One approach to solve this problem is to enable agents to share knowledge. When an agent can utilize the knowledge of other agents through knowledge sharing, the agent can reduce the amount of internal resources that it must use for context acquisition and processing. Furthermore, knowledge sharing can also help agents to acquire additional information that are not directly accessible to them.

An Open and Dynamic Network

A pervasive computing system is usually built on an open and dynamic network. In this network, different agents are built with different types of wired and wireless communication interfaces (Bluetooth, 802.11x, Ethernet, GPRS/GSM, etc.). These agents are also highly dynamic. They dynamically join and leave the network, they dynamically discover other agents to acquire information and services, and they dynamically form teams to achieve goals in a cooperative manner. Knowledge sharing is key part of the agents' behavior in this dynamic environment.

While knowledge sharing is an effective approach to overcome the resource constraint problem of an single agent, it also creates a new set of problems. First, enabling knowledge sharing may add extra complexity to the implementation of an agent. For example, in order to prove the authenticity and determine the reliability of the acquired information, an agent must dedicate sufficient resources for security authentication and trust establishment. When inconsistent information is acquired from multiple sources, an agent must also dedicate sufficient resources for detecting and resolving inconsistent information. Second, in an open environment, not all agents are built to use a common langauge and vocabularies for describing context. Without sharing a common knowledge representation language and ontologies, agents would be unable to share knowledge effectively. Third, as information is acquired through knowledge sharing, some of which may be inconsistent or inaccurate. For this reason, the agents must also dedicate resources to maintain the storage and consistency of this shared knowledge.

Ubiquitous Sensing of Private Information

In a pervasive computing environment, context sensors are typically hidden from the users. Through these sensors, service agents acquire the contextual information about the users and their surrounding environment. They use this knowledge to provide relevant services and information. Sometimes the agents also share the information with each other. Because the users are often focused on the tasks in hand not at the underlying behavior of the surrounding agents, they may not notice exactly how different agents use and share their private information. For this reason, privacy protection is an important issue that must be addressed in the pervasive context-aware systems.

Privacy is about the control of information. The users of a pervasive computing system should be in control of their private information – how their information can be used and who can have access to it. Because it is not feasible to assume users' privacy information can be completely shielded from the ubiquitous sensors and agents, the systems should provide adequate mechanisms to allow users to take control of their privacy. Policy is an effective mechanism for privacy protection [58]. Users can define policy rules to control the use of their private information.

B. Context Broker

CoBrA is a broker-centric agent architecture for supporting context-aware systems in smart spaces. Central to CoBrA is an intelligent agent called *context broker* (see Figure 1). The context broker is a specialized server entity that runs on a resource-rich stationary computer in the space. In a smart space, a context broker has the following responsibilities:

- provide a centralized model of context that all devices, services, and agents in the space can share,
- acquire contextual information from sources that are unreachable by the resource-constrained devices,
- reason about contextual information that cannot be directly acquired from the sensors,
- detect and resolve inconsistent knowledge stored in the shared context model, and
- protect privacy by enforcing policies that users have defined to control the sharing and use of their contextual information.

The functional design of the context broker consists of four modular components. Each component provides distinctive functions for supporting persistent data storage, context reasoning, context acquisition, and privacy protection.

1. *Context knowledge base.* This component manages the storage of the context broker's knowledge. This knowledge includes the ontologies for describing various types of contexts, the ontology instance data

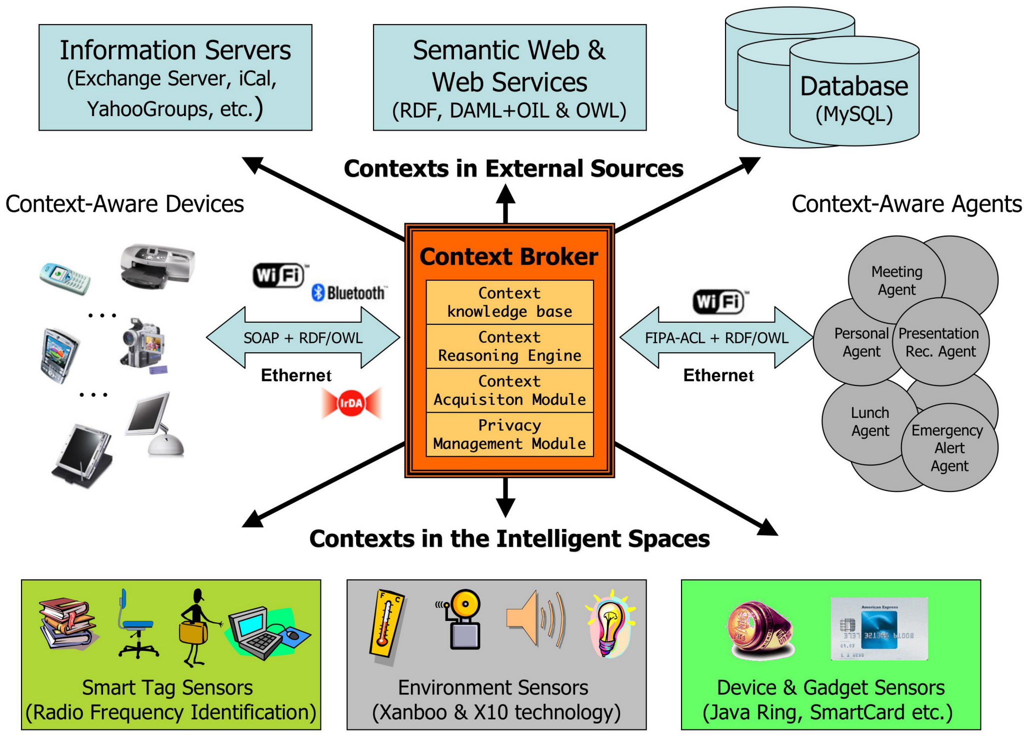


Figure 1: An intelligent context broker acquires context information from devices, agents and sensors in its environment and fuses it into a coherent model, which is then shared with the devices and their agents.

of the acquired contextual information, and the meta-data for describing the storage structure of the represented knowledge.

2. *Context-reasoning engine.* A logical inference engine for reasoning over the acquired contextual information. The function of this engine includes interpreting context based on the acquired sensing data, aggregating the contextual information from multiple sources using ontologies and domain heuristics, detecting and resolving inconsistent information.

3. *Context-acquisition module.* This component is a set of library procedures for acquiring contextual information from sensors, agents, and the Web. It hides the low-level context sensing implementations from the high-level functional components. This middle-ware system design is aimed to improve the reusability of the context sensing procedures.

4. *Privacy-management module.* This component manages the users' privacy policies and controls the sharing of their private information. It is responsible for enforcing the appropriate privacy policies when the context broker attempts to share certain user information. When sharing information, if it is

forbidden by some user's policy, this components also helps to guide the logical reasoning to adjust information granularity.

The above design description only provides a functional specification for implementing a context broker. The developers of a context broker must also consider additional engineering issues when building a context broker for a particular smart space.

C. Implementation Blueprints

In this section, I describe a basic framework for implementing a context broker for a smart meeting room. This framework builds on the context broker design and consists of recommendations for a pragmatic context broker implementation. It addresses the following engineering issues: (i) the development of ontologies for building a context broker, (ii) the service discovery and communication between a context broker and computing entities with heterogenous network interfaces, (iii) the deployment and administration of a context broker, and (iv) the single-point-of-failure scenario in a distributed system.

Note that a number of other issues are not explicitly addressed in this framework, such as network security, data integrity, and scalability. I consider these issues as part of the CoBrA's future works.

How to Develop Ontologies

The use of ontologies in a context broker serves two important purposes. First, it provides an explicit representation of the kind of context information that the context broker is capable of sharing and processing. Second, it helps to disambiguate contextual information that may have different meanings in the distinctive agent implementations.

Ontology development is an iterative process. First, the developers should study the domain in which the context broker is to be deployed – understand what is the essential contextual knowledge that must be represented and understand the appropriate relations and constraints that should be modeled in order to avoid ambiguities. Second, knowing the kind of contextual information that must be represented, the developers should research relevant ontologies that may have been developed in the past and evaluate if these ontologies can be reused. Third, if reusing or extending from the existing ontologies is possible, the developers should define test cases to verify whether or not these ontologies can effectively support the applications of the context broker in the target domain. Otherwise, new ontologies should be developed. Lastly, if the ontologies do not pass all of defined test cases, the developers should again study and re-evaluate the domain applications of the context broker by repeating the first step.

Let's consider a simple exercise example of ontology development. In this example, the goal is to develop a context broker that can acquire user profiles from the Web and share this information with the service agents in a smart meeting room.

First, study the domain applications of the context broker. We discuss with the domain experts of the smart meeting room applications and find out the important concepts that must be represented. Assume that the following is a list of the information that the domain experts have provided: (i) the contact information of a person, (ii) the social network of a person, (iii) the hardware profiles of the devices that a person owns, and (iv) the privacy policies of a person.

Second, research and evaluate whether or not any existing ontologies can be reused or extended. For example, the SOUPA ontology [24] and the FOAF ontology [15] have numerous ontological vocabularies that are suitable for describing people's contact information and social network. The policy ontologies in SOUPA and Rei [58] have defined vocabularies of policy. The FIPA device [42] and CC/PP [64] ontologies have defined typical vocabularies for device hardware and software profiles and Quality of Services (QoS). By either extending or importing from these ontologies, we reduce the engineering effort that is required for ontology development.

Third, develop test cases and verify the feasibility of the ontologies. Assume that a key use of the ontologies in this context broker is to support knowledge sharing (i.e., sharing user profiles with other agents in the system). To verify the feasibility of our ontologies, we define test cases that include example communication messages that resemble the actual communication messages between the context broker and other agents. The developed ontologies pass the test cases if they can be shown to effectively express the content of the defined communication messages.

Lastly, re-evaluate the domain applications if the developed ontologies do not pass all test cases. Assume that the ontologies directly imported from the SOUPA ontology are unable to describe the role changes of a person. We repeat the first step of the ontology development process.

How to Discover and Communicate a Broker

In order to acquire contextual information from a context broker, devices, services, and agents must be able to discover and communicate with the context broker. A fundamental issue is how to enable computing

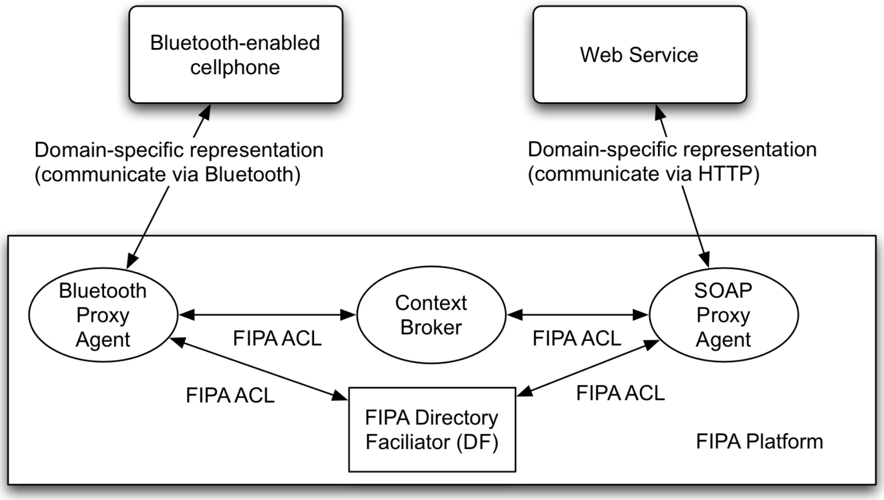

Bluetooth-enabled cellphone
Web Service
Domain-specific representation (communicate via Bluetooth)
Domain-specific representation (communicate via HTTP)
Bluetooth Proxy Agent
Context Broker
SOAP Proxy Agent
FIPA ACL
FIPA ACL
FIPA ACL
FIPA ACL
FIPA Directory Faciliator (DF)
FIPA Platform

Figure 2: Using the proxy design pattern to create surrogate agents to facilitate the discovery and the communication with a FIPA-compliant context broker.

entities with different communication network interfaces to discover and communicate with a context broker. For example, cellphones may have different communication and service discovery interfaces than laptops or agents that run on them. In a smart meeting room, the web services that implement the W3C standards may communicate using protocols that are different from the software agents that implement the FIPA standards.

An approach that addresses this issues is the use of the *proxy* design pattern [46]. The idea is to define surrogates for computing entities with distinctive network interfaces to discover and communicate with a context broker. The following example illustrate the use of the proxy pattern in CoBrA:

Assume that a context broker is implemented as a FIPA agent (i.e., it uses the hosting FIPA platform for publishing services and facilitating communications). The goal of this exercise is to develop a solution that enables cellphones with the Bluetooth interface and web services with the SOAP interface to discover and communicate with the context broker. Using the proxy pattern, we can develop two independent FIPA agents to facilitate the discovery and the communications via Bluetooth and the SOAP protocols, respectively. Figure 2 shows the design diagram of these two agents and the context broker.

How to Deploy and Administer a Broker

For deploying a context broker, a typical configuration is to install the agent on a resource-rich computer in the space. Some tasks associated with the administration of a context broker are similar to those associated with the administration of a typical web service. These tasks include defining service descriptions for supporting service discovery, monitoring the internal behavior of the system via an administrator user interface console.

The following example describes the detail of deploying and administrating a context broker. It is based on a CoBrA prototype that I have developed. The prototype context broker is implemented as a FIPA agent using the JADE programming library. JADE provides a standard FIPA service discovery framework. To register the context broker with a local FIPA Directory Facilitator, the system administrator must define a service registration description, which consists of the ontologies, the content representation languages, and the protocols that are supported by the context broker.

With the emergence of the Semantic Web Services standards, in the future, web service description language such as OWL-S (`http://www.daml.org/services/owl-s/`) could also be used to describe services. Using OWL-S can improve expressiveness and enhance service discovery.

A key reason for monitoring a context broker is to provide the system administrator with a traceable record of the agent's internal behavior. This includes the steps in the execution of logical inferences, the context broker's conversations with other agents, and the changes of the context broker's ontologies and contextual knowledge. Building tools to monitor the context broker, typically there are two approaches: a web-based approach and a stand-alone user interface approach. I recommend the use of the Eclipse platform to develop a stand-alone user interface. Although a web-based approach allows an administrator to perform tasks via a light-weight client (i.e., a standard web browser), but it requires relatively large amount of development overhead. The Eclipse platform is an open platform for tool integration. It allows customized Plug-in tools to be built by reusing the standard Eclipse GUI widgets and UI layouts. I have developed a CoBrA monitoring tool prototype in Eclipse and its details are described in the Chapter VII..

How to Overcome Single-Point-Of-Failure

The centralized design of the context broker could create a "bottle neck" situation in a distributed system. This is also known as the *single-point-of-failure* problem. Because context-aware agents heavily depend on the knowledge of a context broker, in case the context broker fails, the functions of these context-aware agents could be hindered. To overcome the single-point-of-failure problem in CoBrA, I propose a fault-tolerant solution based on the *persistent team* approach described in the Adaptive Agent Architecture [66]. In the rest of this section, I describe the design of this persistent team approach. A reference implementation and the evaluation of the Adaptive Agent Architecture can be found in [66].

The core of the persistent team approach is the notion of a broker team (i.e., a group of broker agents).

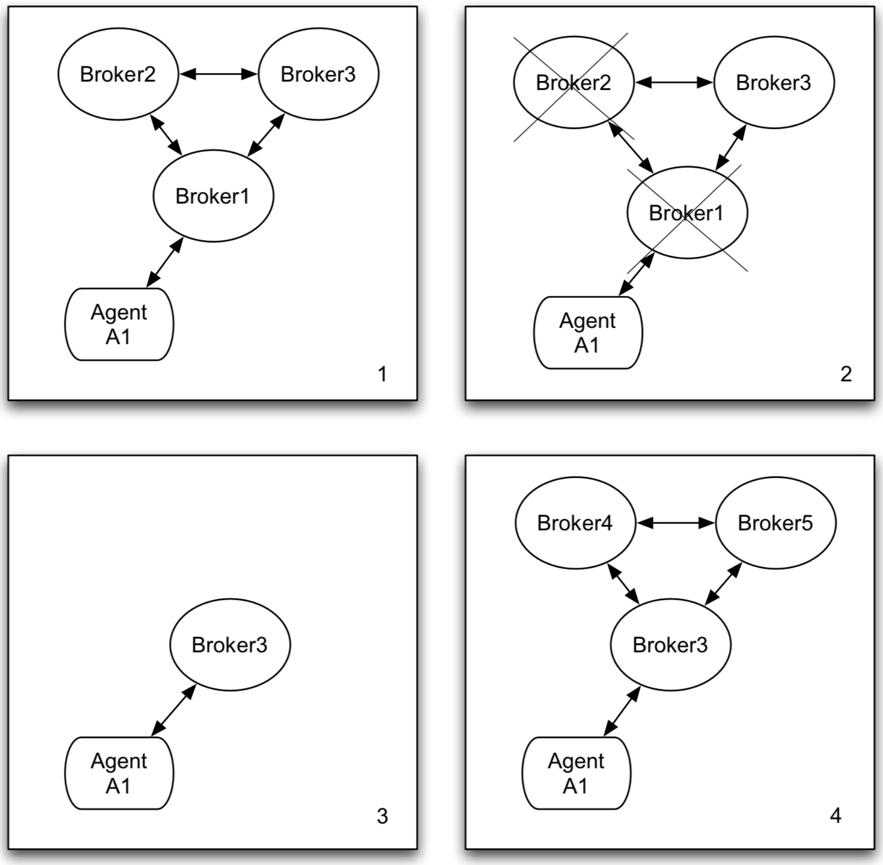


Figure 3: A recovery scenario of a context broker team. (1) Agent A1 is connected to Broker1. (2) Both Broker1 and Broker2 become unavailable, and Agent A1 disconnects from Broker1. (3) The underlying system signals Broker3 about the failure of the other two members, and then "spawns" new team members. (4) Broker3 establishes team intention and team maintenance goal with the new team members, and establishes itself as the new service provider for Agent A1.

This team of brokers share a set of joint commitments. The persistent team approach is an extension of the joint intention theory [30]. In a persistent broker team, the broker agents establish a *team intention* and a *team maintenance goal*, which is informally defined by the following mission statements [66]:

- Whenever an agent registers with the broker team, the brokers have a team intention of connecting with that agent, if it ever disconnects, as long as it remains registered with the team.
- The broker team has a team maintenance goal of having at least N brokers in the team at all times where N is specified during the team formation.

The use of the persistent team approach in CoBrA assumes the following: (i) the members of a broker team are capable of synchronizing their stored contextual knowledge to maintain shared models of the context, (ii) the infrastructure on which the CoBrA is deployed is capable of "spawning" new instances of the context broker when it is necessary to maintain a required number of team members, (iii) the same infrastructure is also capable of detecting the availability of the individual team members, and (iv) all messages send between the broker team members are always delivered in a synchronized manner and will never lost during the network transmission.

The following recovery scenario demonstrates the use of a persistent team: a team of three brokers are deployed in a system (see Figure 3). The team is required to maintain at least two members. When context-aware agent A1 is connected with Broker1, Broker1 has the commitment to notify the rest of the team about A1. Sometime later, due to a system failure in Broker1 and Broker2, they become unavailable and unable to continue to provide contextual information. As the size of the team falls below two, Broker3 signals the underlying infrastructure to spawn additional broker agents. Once the new member agents are created, Broker3 establishes a team intention and team maintenance goal with them. In addition, knowing that agent A1 has previously depended on Broker1 to provide contextual information, Broker3 communicates with A1 and establishes itself as the new service provider. To fulfill its joint commitment, it notifies the rest of the team of this new change.

D. Applications of CoBrA

CoBrA is designed to support pervasive context-aware systems in smart spaces. This section describes three different smart meeting room applications that could be built on CoBrA.

1. Intelligent Personal Agent

An Intelligent Personal Agent (or personal agent for short) is a software agent that maintains personal information for a user. It usually operates on a stationary computer that the user has set up (e.g., on a desktop computer in the office or at home). A personal agent can access the user's private information, such as the daily schedules, address books, personal profiles, location information, etc. It also has the right to decide when and with whom this information can be shared.

Key functions of the personal agent are recording and maintaining a user's context (e.g., what the user is doing, where the user is located, what event the user is attending) and to share this information with other agents that attempt to provide context-aware services to the user. A typical use case of the personal agent is the following:

As the user Alice enters a smart meeting room ITE-201A, the Context Broker of the associated space immediately informs Alice's personal agent. Knowing Alice is located in ITE-201A, the context broker attempts to determine why Alice is there. It asks Alice's personal agent. The personal agent reviews Alice's daily schedule. Without knowing any evidence to the contrary, it concludes that she is scheduled to give a presentation in ITE-201A, and informs the context broker of Alice's speaker role and the URL of her presentation slides. On receiving this information, the context broker shares it with a projector service agent. This agent is permitted by Alice's privacy policy to acquire such information. A few minutes later, the projector service agent downloads the slides and sets up the presentation.

2. Projector Tracking Service

A Projector Tracking Service is a service that monitors the whereabouts of a portable projector. In an office environment, this service can help people to track the location of a public portable projector that has been borrowed for meeting presentations. A public projector is a device that is owned by an organization (e.g., a department) but shared by different people in the organization (e.g., faculty, graduate students).

Key functions of this service are tracking the location of a projector and sending reminders to the user who has borrowed the projector but has not yet returned the device. A typical use case of this service is the following:

As Alice starts to give her PowerPoint presentation, the context broker detects the presence of a projector in the room. Immediately the context broker informs the Projector Tracking Service of the projector location.

Knowing the projector is in the Room ITE-201A, the service asks the context broker who should be responsible for returning the device when the presentation ends. From the meeting schedule, the context broker learns that Alice is invited by Bob, who is the organizer of the meeting. Knowing this information, the context broker sends an SMS message to Bob, inquiring if he is willing to be in charge of the returning of the projector. Bob replies "yes", and the context broker relays this message to the tracking service. To track the location of the projector, the tracking service subscribes to the context broker, requesting to be notified about the device location and the state of the device (i.e., active, sleep, turned off) every 30 minutes.

As the meeting ends, Bob leaves the room and accidentally forgets to return the projector. A couple of hours later, the Projector Tracking Service continuously receives updates about the projector being inactive in the Room ITE-201A. Without having any evidence to the contrary, the service concludes Bob has forgotten to return the device. Immediately, it sends a reminder to Bob asking him to return the projector.

3. Visitor Assistant Service

A Visitor Assistant Service is a service that helps to coordinate and plan typical activities for visiting speakers. In a university, for example, the visiting speakers from other institutions may be unfamiliar with the local environment and require assistants.

Key functions of this service include arranging local transportation, providing directions to various places on the campus, and planning meeting schedules for the visitors. A typical use case scenario of this service is the following:

When the meeting is over, the context broker informs the Visitor Assistant Service that the visiting speaker Alice has finished her presentation. The assistant service asks the context broker for Alice’s schedule, so that it can help to make plans for her. Knowing Alice’s privacy policy permits the schedule information to be shared, the context broker sends this information to the assistant service. According to Alice’s schedule, she has a flight to catch in four hours. Based on this knowledge, the assistant service believes that it should reserve a taxi cab service for Alice getting to the airport. It asks the context broker to get a confirmation from Alice. The context broker sends a request for confirmation message to the Alice via text messaging. Alice replies “yes”. The context broker notifies the assistant service and tells it to go ahead with the reservation.

Since it will be few hours before Alice’s flight departs, the assistant service decides to ask whether Alice would like to arrange a short meeting with some of the graduate students in the department or tour the university campus. The context broker relays this request to Alice. Alice decides that she will tour around the university campus. In particular, she wants to visit the university library. In order to send Alice the directions to the library building, the assistant service needs to know her current location. It asks the context broker. Knowing Alice’s policy permits the location information to be shared at a spatial granularity level that is equal or greater than “building”, the context broker tells the assistant service the building that Alice is currently in. A few seconds later, the walking directions to the library is sent to Alice.

Chapter IV.

ONTOLOGIES

The use of ontologies is an important part of CoBrA. The architecture exploits ontologies expressed in the Web Ontology Langauge OWL for context modeling, context reasoning, and knowledge sharing. Two sets of ontologies are used in CoBrA: the SOUPA ontology and the CoBrA ontology. This chapter describes the details of these ontologies.

A. Background

The ontologies of SOUPA and CoBrA are closely related, but they are designed with distinctive objectives. The CoBrA ontology (COBRA-ONT) is a smart meeting ontology for prototyping CoBrA applications. In particular, the ontology defines the vocabularies for describing the meeting events of the eBiquity Research Group at UMBC. It was first released in March 2003. After the first release, there were five other incremental releases of the revised ontology.

The SOUPA ontology, on the other hand, is a standard ontology for supporting pervasive and ubiquitous computing applications. This SOUPA ontology project was initiated by myself and an international group of researchers who are part of the Semantic Web in UbiComp Special Interest Group (`http://pervasive.semanticweb.org`). Our motivation was to create a standard ontology that brings together many useful ontologies from distinctive domains to support pervasive computing applications. We believe by defining a shared ontology, SOUPA can help developers who are inexperienced in knowledge representation to quickly begin building ontology-driven applications without needing to define ontologies from scratch and to be more focused on the functionalities of the actual system implementations.

At the time of writing this dissertation, SOUPA 2004-06 is the latest release version of the SOUPA ontology, and COBRA-ONT 2004-05 is the latest release version of the CoBrA ontology. The latest version of the CoBrA ontology encourages ontology reuse by importing many top-level concepts from the SOUPA ontology.

The Web Ontology Language OWL

The OWL language is a Semantic Web language for use by computer applications that need to process the content of information instead of just presenting information to humans [75]. This language is developed in part of the Semantic Web initiatives sponsored by the World Wide Web Consortium (W3C).

The current human-centered web is largely encoded in HTML, which focuses largely on how text and images would be rendered for human viewing. Over the past few years we have seen a rapid increase in the use of XML as an alternative encoding, one that is intended primarily for machine processing. The machines which process XML documents can be the end consumers of the information, or they can be used to transform the information into a form appropriate for humans to understand (e.g., as HTML, graphics, and synthesized speech). As a representation language, XML essentially provides a mechanism to declare and use simple data structures, and thus it is not a desirable language for expressing complex knowledge. Enhancements to the basic XML such as XML Schemas address some of the shortcomings, but still do not result in an adequate language for representing and reasoning about the kind of knowledge essential to realizing the Semantic Web vision.

OWL is a knowledge representation language for defining and instantiating ontologies. An ontology is a formal explicit description of concepts in a domain of discourse (or classes), properties of each class describing various features and attributes of the class, and restrictions on properties [77].

The normative OWL exchange syntax is RDF/XML. Ontologies expressed in OWL are usually placed on web servers as web documents, which can be referenced by other ontologies and downloaded by applications that use ontologies. I refer to these web documents as *ontology documents*.

Related Ontologies

Part of the SOUPA vocabularies are adopted from a number of different consensus ontologies. The strategy for developing SOUPA is to borrow terms from these ontologies but not to import them directly. Although the semantics for importing ontologies is well defined [5], by choosing not to use this approach we can effectively limit the overhead in requiring reasoning engines to import ontologies that may be irrelevant to pervasive computing applications. However, in order to allow better interoperability between the SOUPA applications and other ontology applications, many borrowed terms in SOUPA are mapped to the foreign ontology terms using the standard OWL ontology mapping constructs (e.g., `owl:equivalentClass` and `owl:equivalentProperty`).

The ontologies that are referenced by SOUPA include the Friend-Of-A-Friend ontology (FOAF) [15, 88], DAML-Time and the entry sub-ontology of time [52, 79], the spatial ontologies in OpenCyc [71], Regional Connection Calculus (RCC) [90], COBRA-ONT (i.e., COBRA-ONT 2003-11) [21], MoGATU BDI ontology [82], and the Rei policy ontology [57].

The FOAF ontology allows the expression of personal information and relationships, and is a useful building block for creating information systems that support online communities [38]. Pervasive computing applications can use the FOAF ontologies to express and reason about a person's contact profile and social connections to other people in their close vicinity.

The vocabularies of the DAML-Time ontology and the entry sub-ontology of time are designed for expressing temporal concepts and properties common to any formalization of time. Pervasive computing applications can use these ontologies to share a common representation of time and to reason about the temporal orders of different events.

The OpenCyc spatial ontologies define a comprehensive set of vocabularies for symbolic representation of space. The ontology of RCC consists of vocabularies for expressing spatial relations for qualitative spatial reasoning. In pervasive computing applications, these ontologies can be exploited for describing and reasoning about location and location context [21].

Both COBRA-ONT and MoGATU BDI ontologies are aimed for supporting knowledge representation and ontology reasoning in pervasive computing environment. The design of COBRA-ONT focuses on modeling context in smart meeting rooms [21], and the design of MoGATU BDI ontology focuses on modeling the belief, desire, and intention of human users and software agents [82].

The Rei ontology defines a set of deontic concepts (i.e., rights, prohibitions, obligations, and dispensations) for specifying and reasoning about security access control rules. In a pervasive computing environment, users can use this policy ontology to specify high-level rules for granting and revoking the access rights to and from different services [60].

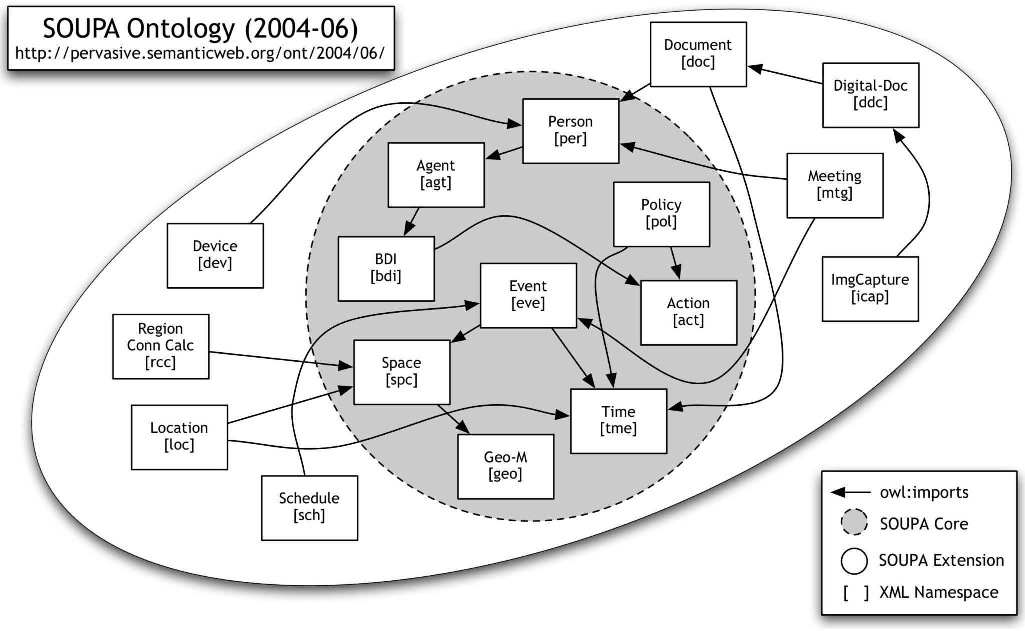


Figure 4: SOUPA consists of two sets of ontology documents: SOUPA Core and SOUPA Extension. The OWL `owl:imports` construct is used to enable a modular design of the ontology. Different domain vocabularies are grouped under different XML namespaces.

B. SOUPA Ontology

SOUPA consists of two distinctive but related set of ontologies: SOUPA Core and SOUPA Extension. The set of the SOUPA Core ontologies attempts to define generic vocabularies that are universal for different pervasive computing applications. The set of SOUPA Extension ontologies, extended from the core ontologies, define additional vocabularies for supporting specific types of applications and provide examples for the future ontology extensions.

Note that the structure of the ontology merely suggests certain vocabularies are more general than the others in supporting pervasive computing applications, and there is no inherent computational complexity difference in adopting either set of the ontologies.

1. SOUPA Core

The SOUPA Core ontologies consist of vocabularies for expressing concepts that are associated with person, agent, belief-desire-intention (BDI), action, policy, time, space, and event. The ontologies are grouped into nine distinctive ontology documents. Figure 4 shows a diagram of the ontology documents and their associated relations.

Person

This ontology defines typical vocabularies for describing the contact information and the profile of a person. The OWL class `per:Person` is defined to represent a set of all people in the SOUPA domain, and is equivalent to the `foaf:Person` class in the FOAF ontology (i.e., the `owl:equivalentClass` property holds between the `per:Person` and `foaf:Person` class). An individual of the class can be described by a set of properties, which include basic profile information (name, gender, age, birth date, etc.), the contact information (email, mailing address, homepage, phone numbers, instant messaging chat ID, etc.), and social and professional profile (people that a person is friend of, organizations that a person belongs to). In addition, all property vocabularies that are applicable to describe a person in the FOAF ontology can also be used to describe an individual of the `per:Person` class. This is because all individuals of the `per:Person` class are also individuals of the `foaf:Person` class. The following shows a partial ontology description of the person Harry Chen:

```
<per:Person>
  <per:firstName rdf:datatype="&xsd;string>Harry</per:firstName>
  <per:lastName rdf:datatype="&xsd;string>Chen</per:lastName>
  <per:gender rdf:resource="&per;Male"/>
  <per:birthDate rdf:datatype="&xsd;date">1976-12-26</per:birthDate>

  <per:homepage rdf:resource="http://umbc.edu/people/hchen4"/>
  <foaf:weblog rdf:resource="http://umbc.edu/people/hchen4"/>

  <per:hasSchoolContact rdf:resource="#SchoolContact"/>
  <per:hasHomeContact rdf:resource="#HomeContact"/>

  <foaf:workplaceHomepage rdf:resource="http://ebiquity.umbc.edu"/>
  <foaf:workplaceHomepage rdf:resource="http://www.umbc.edu"/>
  <foaf:workplaceHomepage rdf:resource="http://www.cs.umbc.edu"/>
</per:Person>

<per:ContactProfile rdf:ID="SchoolContact">
  <per:address rdf:datatype="&xsd;string">
    Dept. of CSEE, UMBC, 1000 Hilltop Circle, Baltimore, MD 21250, USA
   </per:address>
   <per:phone rdf:datatype="&xsd;string>+1-410-455-8648</per:phone>
   <per:email rdf:resource="mailto:harry.chen@umbc.edu"/>
   <per:im rdf:resource="aim:goim?screenname=hc1379"/>
</per:ContactProfile>

<per:Email rdf:about="mailto:harry.chen@umbc.edu"/>
<per:Homepage rdf:about="http://www.aim.com"/>
<per:ChatID rdf:about="aim:goim?screenname=hc1379">
  <per:providedBy rdf:resource="http://www.aim.com"/>
</per:ChatID>

<per:ContactProfile rdf:ID="HomeContact">
  ...
</per:ContactProfile>

<foaf:knows>
  <foaf:Person>
     <foaf:name>Tim Finin</foaf:name>
     <foaf:mbox_sha1sum>49953...148d37</foaf:mbox_sha1sum>
  </foaf:Person>
</foaf:knows>
</rdf:RDF>
```

Policy & Action

Security and privacy are two growing concerns in developing and deploying pervasive computing systems [16, 62, 47]. Policy is an emerging technique for controlling and adjusting the low-level system behaviors by specifying high-level rules [35].

The SOUPA policy ontology defines vocabularies for representing security and privacy policies and a description logic based mechanism for reasoning about the defined policies. The defined vocabularies in this ontology are influenced by the Rei policy language [57].

A policy is a set of rules that is specified by a user or a computing entity to restrict or guide the execution of actions. For example, in the context of system security, a system administrator may use policies to define who has the right to execute what services; in the context privacy protection, a user may use policies to restrict the type of personal information that can be shared by the public services.

The ontology representation of an action is defined in the `action` ontology document. The class `act:Action` represents a set of all actions. Individuals of this class can have a set of property values, which include (i) `act:actor` – the entity that performs the action, (ii) `act:recipient` – the entity that receives the effect after the action is performed, (iii) `act:target` – the object that the action applies to, (iv) `act:location` – the location at where the action is performed, (v) `act:time` – the time at which the action is performed, (vi) `act:instrument` – the thing that the actor uses to perform the action.

The following shows a partial ontology that defines a special class of knowledge sharing action:

```
<owl:Class rdf:ID="ShareHarryLocInfoWithEBMembers">
    <owl:intersectionOf rdf:parseType="Collection">
       <owl:Class rdf:about="&act;Action"/>

       <owl:Restriction>
         <owl:onProperty rdf:resource="&act;actor">
         <owl:hasValue>
            <agt:Agent rdf:about="ctb@cobra1.cs.umbc.edu"/>
         </owl:hasValue>
       </owl:Restriction>

       <owl:Restriction>
          <owl:onProperty rdf:resource="&act;target"/>
          </owl:allvaluesFrom rdf:resource="#LocationContextOfHarry"/>
       </owl:Restriction>

       <owl:Restriction>
          <owl:onProperty rdf:resource="&act;recipient"/>
          <owl:allValuesFrom rdf:resource="&eb;EbiquityMembers"/>
       </owl:Restriction>
    </owl:intersectionOf>
</owl:Class>

<owl:Class rdf:ID="LocationContextOfHarry">
   <owl:intersectionOf rdf:parseType="Collection">
      <owl:Class rdf:about="&loc;LocationContext"/>
      <owl:Restriction>
         <owl:onProperty rdf:resource="&loc:locationContextOf"/>
         <owl:hasValue>
            <per:Person rdf:about="http://umbc.edu/people/hchen4"/>
         </owl:hasValue>
      </Restriction>
   </owl:intersectionOf>
</owl:Class>
```

The above example describes a set of actions of which the actor is agent `ctb@cobra1.cs.umbc-.edu`, the recipient is any member of the eBiquity group, the target, or the information to be shared, is Harry's location information.

In SOUPA, a policy consists of rules that either *permit* or *forbid* the execution of certain described actions. Defined in the `policy` ontology document, the `pol:Policy` class represents a set of all policies. For a given policy individual, it may be associated with one or more `pol:permits` or `pol:forbids` properties. The range of these two properties are the `pol:PermittedAction` class and the `pol:Forbidden-Action` class, respectively.

The following example shows a policy that gives agent `ctb@cobra1.cs.umbc.edu` the permission to share Harry's location information with all eBiquity members:

```
<pol:Policy rdf:about="&cobra;harrychen-policy">
  <pol:policyOf>
    <per:Person rdf:about="http://umbc.edu/people/hchen4">
       <per:name rdf:datatype="&xsd;string">Harry Chen</per:name>
    </per:Person>
  </pol:policyOf>

  <pol:defaultPolicyMode rdf:resource="&pol;RequiresExplicitPermission"/>

  <pol:permits rdf:resource="#ShareHarryLocInfoWithEBMembers"/>
</pol:Policy>
```

The policy ontology also defines vocabularies for describing meta information about individual policies. This information includes the author of a policy (`pol:creator`), the entity that enforces a policy (`pol:enforcer`), the creation time of a policy (`pol:createdOn`), and the default reasoning mode of a policy (`pol:defaultPolicyMode`).

The design of the SOUPA policy exploits *classification* as a means to reason about policies. A typical process flow of the system implementation is the following: (i) a user or a system administrator defines a policy, (ii) the policy is transmitted to the appropriate policy enforcer (e.g., a security or a privacy protection agent), (iii) before the policy enforcer can permit other agents to perform an action, it creates an explicit representation of the action using the SOUPA action ontology, (iv) this represented action is then loaded into a description logic reasoner (e.g., Racer [48] or FaCT [53]) along with the associated ontology, and (v) the policy enforcer will permit the execution of the action if the action is classified as type of `pol:Permitted-Action`, and it will forbid the execution of the action if the action is classified as type of `pol:Forbidden-Action`.

In case if an input action is classified as both `pol:PermittedAction` and `pol:Forbidden-Action`, then the policy enforcer will report there is an inconsistency in the policy, and may forbid the execution of the action by default. In case if the action cannot be classified as either class, the policy enforcer will decide whether the action should be permitted or forbidden based on the default policy mode (see the above example). If the mode is `pol:RequiresExplicitPermission`, then the action will be forbidden. If the mode is `pol:RequiresNoExplicitPermission`, then the action will be permitted.

Agent & BDI

When building intelligent pervasive computing systems, sometimes it is useful to model computing entities as *agents* [109]. In SOUPA, agents are defined with a strong notion of agency [109], which is characterized by a set of *mentalistic* notions such as knowledge, belief, intention, and obligation. In this ontology, both computational entities and human users can be modeled as agents.

When the goals, plans, desires, and beliefs of different agents are explicitly represented in the ontologies, this information can help independently developed agents to share a common understanding of their "mental" states, helping them to cooperate and collaborate. The explicitly represented human user's mental states can help computing agents to reason about the specific needs of the users in a pervasive environment.

Two ontology documents are related to this ontology: `agent` and `bdi`. The `agt:Agent` class represents a set of all agents in the SOUPA domain. It has three properties that characterize an agent's "mental" state: `agt:believes`, `agt:desires`, and `agt:intends`. The respective range values of these properties are the `bdi:Fact`, `bdi:Desire`, and `bdi:Intention` classes. The goals of an agent are considered to be a special type of desire, which is expressed by defining the `agt:hasGoal` property as a sub-property of the `agt:desires` property.

The `bdi:Fact` class is a subclass of the `rdf:Statement` class, which represents a set of reified RDF statements [14]. A reified RDF statement consists of the `rdf:subject`, `rdf:object`, and `rdf:predicate` properties.

The `bdi:Desire` class defines a set of world states that agents desire to bring about. Every instances of this class can be characterized by the property `bdi:endState`. The range restriction of this property is unspecified in the `bdi` ontology document. Application developers are responsible for defining the representation of different world states. Some suggested representations are (i) symbolic names, e.g., a set of pre-defined RDF resource URI and (ii) meta-representation, e.g., each world state description is a set of reified RDF statements.

The `bdi:Intention` class represents a set of plans that agents intend to execute. Plans are defined in terms of actions, pre-conditions, and effects. The `bdi:Plan` class is defined as a subclass of the `act:Action` class with additional properties, namely `bdi:preCondition` and `bdi:effect`.The representation of pre-conditions and effects are unspecified in this ontology, and it is left to be defined by the application ontologies.

Sometimes it may be useful to describe whether or not different desires of an agent are in conflict of each other, and whether or not certain desires are achievable. The cause of desire conflicts may be due to inconsistent beliefs in the knowledge base, conflicting user preferences, or conflicting systems policies. The cause of unachievable desires may be due to the change of situational conditions. In the `bdi` ontology document, different subclasses of the `bdi:Desire` class, `bdi:ConflictingDesire`, `bdi:NonConflicting-Desire`, `bdi:AchievableDesire`, and `bdi:UnachievableDesire`, are defined for classifying different types of agent desires.

Time

SOUPA defines a set of ontologies for expressing time and temporal relations. They can be used to describe the temporal properties of different events that occur in the physical world.

Part of the SOUPA ontology adopts the vocabularies of the DAML-time and the entry sub-ontology of time. The basic representation of time consists of the `tme:TimeInstant` and `tme:TimeInterval` classes. All individual members of these two classes are also members of the `tme:TemporalEntity` class, which is an OWL class that is defined by taking the union of the `tme:TimeInstant` and `tme:Time-Interval` classes. The set of all temporal things that are divided into two disjoint classes: `tme:Instant-Thing`, things with temporal descriptions that are type of time instant, and `tme:IntervalThing`, things with temporal descriptions that are type of time interval. The union of these two classes forms the `tme:-TemporalThing` class.

In order to associate temporal things with date/time values (i.e., their temporal descriptions), the `tme:at` property is defined to associate an instance of the `tme:InstantThing` with an XML `xsd:dateTime` datatype value (e.g., 2004-12-25T12:32:12), and the `tme:from` and `tme:to` properties are defined to associate an instance of the `tme:IntervalThing` with two different `tme:TimeInstant` individuals. The following example shows the representation of a time interval with the associated temporal description:

```
<tme:TimeInterval>
```

```
  <tme:from>
    <tme:TimeInstant>
      <tme:at rdf:datatype="xsd;dateTime">2004-02-01T12:01:01</tme:at>
    </tme:TimeInstant>
  </tme:from>
  <tme:to>
    <tme:TimeInstant>
      <tme:at rdf:datatype="xsd;dateTime">2004-02-11T13:41:21</tme:at>
    </tme:TimeInstant>
  </tme:to>
</tme:TimeInterval>
```

For describing the order relations between two different time instants, the ontology defines the following properties: `tme:before`, `tme:after`, `tme:beforeOrAt`, `tme:afterOrAt`, and `tme:sameTimeAs`. Both `tme:before` and `tme:after` properties are defined of type `owl:TransitiveProperty`. The `tme:sameTimeAs` property expresses that two different time instants are associated with equivalent date/time values and is defined of type `owl:SymmetricProperty`.

For describing the order relations between two different temporal things (i.e., time instants and time intervals), the ontology defines the following properties: `tme:startsSoonerThan`, `tme:startsLaterThan`, `tme:startsSameTimeAs`, `tme:endsSoonerThan`, `tme:endsLaterThan`, `tme:endsSameTimeAs`, `tme:startsAfterEndOf`, and `tme:endsBeforeStartOf`. The first three properties respectively express that for any two given temporal things A and B, the starting time of A is before the starting time of B, the starting time of A is after the starting time of B, and the starting time of A is the same as the starting time of B. The next three properties respectively express that for any two given temporal things A and B, the ending time of A is before the ending time of B, the ending time of A is after the ending time of B, and the ending time of A is the same as the ending time of B. The `tme:startsAfterEndOf` property expresses that the beginning of one temporal thing is after the ending of another temporal thing, and the `tme:endsBeforeStartOf` property expresses the inverse of this property.

Space

This ontology is designed to support reasoning about the spatial relations between various types of geographical regions, mapping from the geo-spatial coordinates to the symbolic representation of space and *vice versa*, and the representation of geographical measurements of space. Part of this ontology vocabularies are adopted from the spatial ontology in OpenCyc and the OpenGIS vocabularies [32].

Two ontology documents are related to this ontology: `space` and `geo-measurement`. The first ontology document defines a symbolic representation of space and spatial relations, and the second document defines typical geo-spatial vocabularies (e.g., longitude, latitude, altitude, distance, and surface area).

In the symbolic representation model, the `spc:SpatialThing` class represents a set of all things that have spatial extensions in the SOUPA domain. All spatial things that are typically found in maps or construction blueprints are called `spc:GeographicalSpace`. This class is defined as the union of the `spc:-GeographicalRegion`, `spc:FixedStructure`, and `spc:SpaceInAFixedStructure` classes.

An individual member of the `spc:GeographicalRegion` class typically represents a geographical region that is controlled by some political body (e.g., the country US is controlled by the US government). This relation is expressed by the `spc:controls` property, the domain of which is `spc:GeopoliticalEntity` and the range of which is `spc:GeographicalRegion`. Knowing which political entity controls a particular geographical region, a pervasive computing system can choose to apply the appropriate policies defined by the political entity to guide its behavior. For example, a system may apply different sets of privacy protection schemes based on the policies defined by the local political entities.

To support spatial containment reasoning, individual members of the `spc:GeographicalSpace` class can relate to each other through the `spc:spatiallySubsumes` and `spc:spatiallySubsumedBy` properties. For example, a country region may spatially subsume a state region, a state region may spatially subsume a building, and a building may spatially subsume a room. Knowing the room in which a device is located, we can infer the building, the state and the country that spatially subsume the room.

In the geo-spatial representation model, the individual members of the `spc:SpatialThing` class are described by location coordinates (i.e., longitude, latitude, and altitude). This relation is expressed by the `spc:hasCoordinates` property, the range of which is the `geo:LocationCoordinates` class. In this model, multiple location coordinates can be mapped to a single geographical region (e.g., a university campus typically covers multiple location coordinates.). This relation is useful for defining spatial mapping between different geographical locations and GPS coordinates. This information can enable a GPS-enabled device to query the symbolic representation of its present location for a given set of longitude, latitude, and altitude.

Event

Events are event activities that have both spatial and temporal extensions. An event ontology can be used to describe the occurrence of different activities, schedules, and sensing events. In the `event` ontology document, the `eve:Event` class represents a set of all events in the domain. However, the definition of this class is silent about its temporal and spatial properties.

The `eve:SpatialTemporalThing` class represents a set of things that have both spatial and temporal extensions, and it is defined as the intersection of the `tme:TemporalThing` and `spc:SpatialThing` classes. To specifically describe events that have both temporal and spatial extensions, `eve:SpatialTemporalEvent` class is defined as the intersection of the `eve:SpatialTemporalThing` and `eve:Event` classes.

The following example shows how the ontology can be used to describe an event in which a Bluetooth device has been detected on 2004-02-01 at 12:01:01 UTC, and the event occurs at a location that is described by longitude -76.7113 and latitude 39.2524:

```
<owl:Class rdf:ID="DetectedBluetoothDev">
  <rdfs:subClassOf rdf:resource="&eve;TemporalSpatialEvent"/>
</owl:Class>

<owl:ObjectProperty rdf:ID="foundDevice">
  <rdfs:domain rdf:resource="#DetectedBluetoothDev"/>
</owl:ObjectProperty>

<DetectedBluetoothDev>
  <spc:hasCoordinates>
    <geo:LocationCoordinates>
      <geo:longitude rdf:datatype="&xsd;string">-76.7113</geo:longitude>
      <geom:latitude rdf:datatype="&xsd;string">39.2524</geom:latitude>
    </geo:LocationCoordinates>
  </spc:hasCoordinates>

  <foundDevice rdf:resource="url-x-some-device"/>
  <tme:at>
    <tme:TimeInstant>
      <tme:at rdf:datatype="xsd;dateTime">2004-02-01T12:01:01</tme:at>
    </tme:TimeInstant>
  </tme:at>
<DetectedBluetoothDev>
```

2. SOUPA Extension

The SOUPA Extension ontologies are created for two purposes: (i) define an extended set of vocabularies for supporting specialized domains of pervasive computing, and (ii) demonstrate how to define new ontologies that extend the SOUPA Core ontologies. At present, the SOUPA Extension consists of experimental ontologies for supporting pervasive context-aware applications in smart spaces and peer-to-peer data management in a pervasive computing environment.

- **Meeting & Schedule.** For describing typical information associated with meetings, event schedules, and event participants. They can help smart meeting systems to represent and reason about the context of a meeting, e.g., do all meeting attendees currently located in the meeting room, and what is the end time of this meeting?

- **Document & Digital Document.** For describing meta-information about documents and digital documents, e.g., the creation date and the author of a document, the source URL of a digital document, file size, and file type.

- **Image Capture.** When a camera phone takes a picture, this event is type of image capturing event. This ontology defines vocabularies for describing image capturing events, e.g., where and when was the picture taken, and which device has taken the picture?
- **Region Connection Calculus.** A spatial ontology that supplements the core space ontology. Based on the Region Connection Calculus [90], this ontology defines vocabularies for expressing spatial relations for qualitative spatial reasoning.
- **Location.** For describing sensed location context of a person or an object. The location context is information that describes the whereabouts of a person or an object, which includes both temporal and spatial properties.

C. CoBrA Ontology

The CoBrA ontology extends the SOUPA ontology and defines the basic ontology vocabularies for prototyping CoBrA applications. The COBRA-ONT aimed to support smart meeting room applications for the eBiquity group meetings at UMBC. The ontology covers typical concepts associated with UMBC geographical information, eBiquity group meetings, and actions performed by the smart meeting applications. The full ontology documents are accessible at `http://cobra.umbc.edu/ont/2004/05`.

1. eBiquity Geo-Spatial Ontology

This ontology defines vocabularies for modeling certain physical places located on the UMBC campus and their spatial relations and constraints. In particular, it defines ontology classes for symbolic representations of rooms, buildings, campus, states, and countries. It also defines instants of these geo-spatial classes and the associated relations.

Sometimes it is useful to map a symbolic representation of a physical place to a geometric representation. For example, the UMBC Main Campus has a symbolic representation that is a class individual of the `UniversityCampus` class. To add geometric representation to the same class individual, we use the `spc:hasCoordinates` property.

```
<UniversityCampus rdf:ID="UMBCMainCampus">
  <spc:spatiallySubsumes rdf:resource="#ITE"/>
```

```
    <spc:hasCoordinates>
      <geom:LocationCoordinates rdf:about="campusCoordinates">
        <geom:latitude rdf:datatype="&xsd;string">39.2524</geom:latitude>
        <geom:longitude rdf:datatype="&xsd;string">-76.7113</geom:longitude>
      </geom:LocationCoordinates>
    </spc:hasCoordinates>
  </UniversityCampus>
```

2. eBiquity Meeting Ontology

The eBiquity Research Group at UMBC usually hosts group meetings every week. During the weekly meetings, the group members or invited speakers often give presentations. This ontology extends the SOUPA meeting ontology and schedule ontology to model the eBiquity weekly meetings.

Key concepts covered by the ontology include the modeling of the eBiquity group membership, the friends of the eBiquity group members, and the meeting context. The meeting context consists of descriptions about the speaker of the presentation, the meeting organizer, the meeting attendees, presentation video file, event photos, and voice recording of the discussions.

```
  <owl:Class rdf:ID="EbiquityMeeting">
    <rdfs:subClassOf>
      <owl:Class rdf:about="&mtg;Meeting"/>
    </rdfs:subClassOf>

    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#speaker"/>
        <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
      </owl:Restriction>
    </rdfs:subClassOf>

    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="&spc;location"/>
        <owl:hasValue rdf:resource="&ebgeo;ITE325B"/>
      </owl:Restriction>
    </rdfs:subClassOf>

    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="&mtg;organizedBy"/>
        <owl:hasValue rdf:resource="#timfinin"/>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>

  <per:Person rdf:ID="timfinin">
     <per:name rdf:datatype="&xsd;string">Tim Finin</per:name>
     <per:workplaceHomepage rdf:resource="http://ebiquity.umbc.edu"/>
     <per:workplaceHomepage rdf:resource="http://www.cs.umbc.edu"/>
  </per:Person>
```

3. eBiquity Action Ontology

This ontology is aimed to support privacy protection in a context broker. It extends the SOUPA action ontology. In CoBrA, policies are used to protect privacy by controlling the sharing of information. Sharing information among the agents involves communication. This ontology defines the communication vocabularies between a context broker and other agents.

The OWL class `ebact:Communication` represents a set of all actions that involve communications between two agents. Property constraints are defined on the set of actions properties that inherited from the `act:Action` class.

```
<owl:Class rdf:ID="Communicate">
     <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="&act;Action"/>

        <owl:Restriction>
           <owl:onProperty rdf:resource="&act;actor"/>
           <owl:allValuesFrom>
              <owl:Class>
                 <owl:unionOf rdf:parseType="Collection">
                    <owl:Class rdf:about="&per;Person"/>
                    <owl:Class rdf:about="&agt;Agent"/>
                 </owl:unionOf>
              </owl:Class>
           </owl:allValuesFrom>
        </owl:Restriction>

        <owl:Restriction>
           <owl:onProperty rdf:resource="&act;actor"/>
           <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
        </owl:Restriction>

        <owl:Restriction>
           <owl:onProperty rdf:resource="&act;recipient"/>
           <owl:allValuesFrom>
              <owl:Class>
                 <owl:unionOf rdf:parseType="Collection">
                    <owl:Class rdf:about="&per;Person"/>
                    <owl:Class rdf:about="&agt;Agent"/>
                 </owl:unionOf>
              </owl:Class>
           </owl:allValuesFrom>
        </owl:Restriction>

        <owl:Restriction>
           <owl:onProperty rdf:resource="&act;recipient"/>
           <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
        </owl:Restriction>

        <owl:Restriction>
           <owl:onProperty rdf:resource="&act;target"/>
           <owl:someValuesFrom rdf:resource="#Information"/>
        </owl:Restriction>
     </owl:intersectionOf>
  </owl:Class>
```

To describe a specific type of communication action (e.g. the communications between a specific context broker and other agents), additional subclasses are defined (e.g., `ebact:BrokerCommunicationAction`. The subclasses define additional restricts on the inherited properties. In the following example, the class is defined a communication action that involves an eBiquity context broker being the actor of the action:

```
  <owl:Class rdf:ID="BrokerCommunicateAction">
     <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Communicate"/>

        <owl:Restriction>
           <owl:onProperty rdf:resource="&act;actor"/>
           <owl:allValuesFrom rdf:resource="&ebm;ContextBroker"/>
        </owl:Restriction>

        <owl:Restriction>
           <owl:onProperty rdf:resource="&act;actor"/>
           <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
        </owl:Restriction>
     </owl:intersectionOf>
  </owl:Class>
```

Chapter V.

CONTEXT REASONING

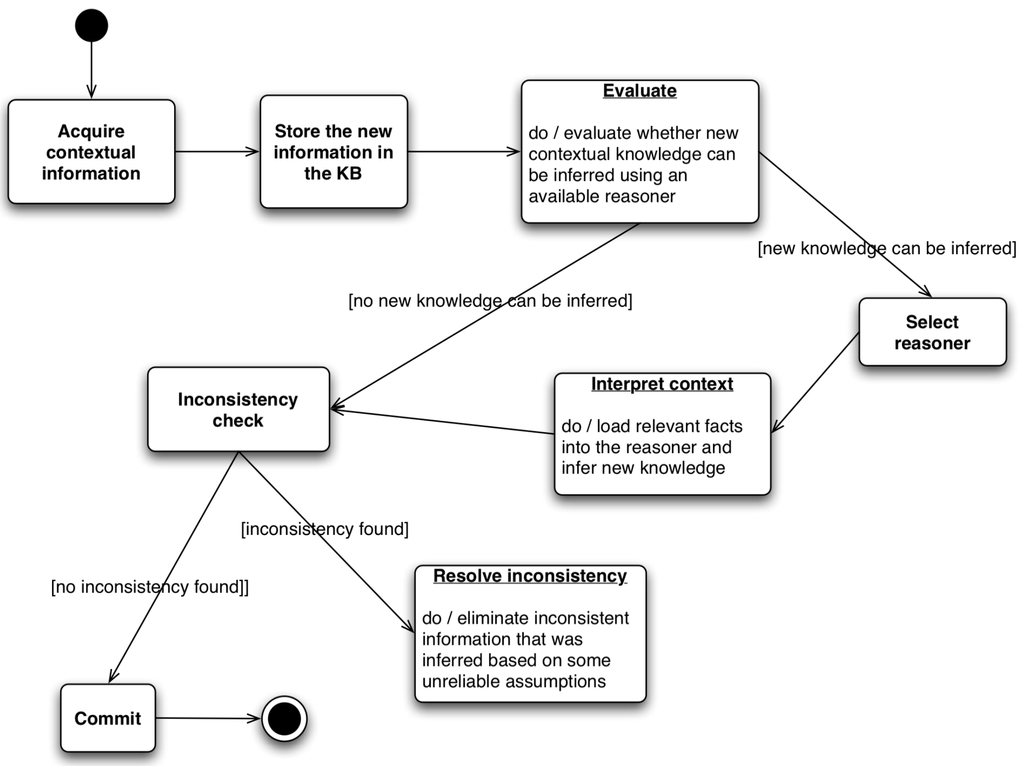
CoBrA supports two kinds of context reasoning: (i) reasoning for building a shared model of context, and (ii) reasoning for maintaining a consistent model of context. The first kind of reasoning involves the use of inference to interpret sensing information, and the other kind of reasoning involves the use of inference to detect and resolve inconsistent information. This chapter describes the context reasoning components of CoBrA.

A. The Need for a Rule-based Framework

Context reasoning is essential to every context-aware systems. The function of a context-aware system depends on the support of context reasoning. In the previous systems [94, 36, 4, 29], context reasoning is typically implemented as procedures using programming languages such Java and C++. These object-oriented languages encourages code reuse at the API level. However, they lack expressive power for knowledge representation. When context reasoning is implemented using these languages, the non-declarative representation of the reasoning logic often makes code modification and human inspection difficult.

To address these issues, I chose to design CoBrA's context reasoning with a rule-based logical inference approach. The advantages of this approach are as follows.

- An explicit representation of the context reasoning rules can help to separate the high-level reasoning logic from the low-level functional implementation. By separating the logic from the functional implementation, developers can modify or replace context reasoning components without requiring significant amount of re-programming efforts.



Acquire contextual information
Store the new information in the KB
Evaluate
do / evaluate whether new contextual knowledge can be inferred using an available reasoner
[new knowledge can be inferred]
[no new knowledge can be inferred]
Select reasoner
Inconsistency check
Interpret context
do / load relevant facts into the reasoner and infer new knowledge
[inconsistency found]
[no inconsistency found]]
Resolve inconsistency
do / eliminate inconsistent information that was inferred based on some unreliable assumptions
Commit

Figure 5: A UML diagram that depicts the state transitions of CoBrA's context reasoning framework.

- A rule-based approach allows many well-defined logic models of general concepts such as time and space to be directly mapped into the context reasoning implementation. The logical inference associated with these general concepts are often useful in context reasoning [24].

- When context interpretation rules are explicitly represented, meta-reasoning techniques can be developed to detect and resolve inconsistent context interpretations.

B. System Design and Implementation

CoBrA's context reasoning framework comprises a set of state transitions (see Figure 5). When a context broker acquires contextual information from the external sources, it stores the information in a persistent knowledge base. This assertion of new knowledge triggers the context broker's reasoning behavior. Depending on the type of the asserted knowledge, different rule-based logical inferences are executed to interpret the context. Before any new knowledge can be committed in the knowledge base, inconsistency checks are performed. If conflicting information is detected, the appropriate reasoning behaviors are triggered to resolve this inconsistency.

Based on this context reasoning framework, I developed a context broker prototype to demonstrate its feasibility in supporting context reasoning. In this prototype, the context broker is implemented as a FIPA-compliant agent using the JADE programming API.

The functional behavior of a context broker is implemented as a set of JADE agent behavior classes. These behavior classes share a common set of the agent's resources. These include a shared knowledge base, a collection of context reasoners, and the essential class objects for engaging in agent communications and service discovery in a FIPA platform. At runtime, the execution of these behavior classes are managed by an internal scheduler of the JADE agent super class (i.e., the context broker agent class is a subclass of the JADE agent class).

The communication between a context broker and other agents follows the FIPA Agent Communication standards [44]. The implementation defines several interaction protocols for agents to communicate with a context broker. These protocols are communication protocols for informing a context broker about certain situational conditions, querying a context broker for certain contextual knowledge, and convincing a context broker to believe certain knowledge is true through argumentation [73].

The context broker implementation supports two different types of content message representations. Depending on the purpose of communication, content messages are represented using different languages. For sharing information with other agents, an XML/RDF representation of the OWL language is used. For answering queries about certain contextual knowledge, the RDQL language [98] is used. The ontological vocabularies used in these representations are imported from the SOUPA and CoBrA ontologies.

The SOUPA and CoBrA ontologies are also used in expressing the internal knowledge of a context broker. The knowledge of a context broker is represented as RDF statements and is stored in a persistent knowledge base. This knowledge base is backed by a MySQL database. The implementation exploits the Jena API to manage the knowledge base and to provide access methods to the knowledge base. Figure 6 shows a UML diagram of the context broker implementation.

The CoBrA's context reasoning uses a number of different rule-based systems, which include the Jena rule-based reasoners [92], JESS (Java Expert System Shell) [45], and Theorist (an assumption-based reasoner in Prolog) [87]. Different rule-based systems provide different logical inference support for context reasoning. The Jena rule-based reasoners are used for OWL ontology inferences, the JESS rule engine is used for interpreting context using domain specific rules, and the Theorist system is used for supporting the

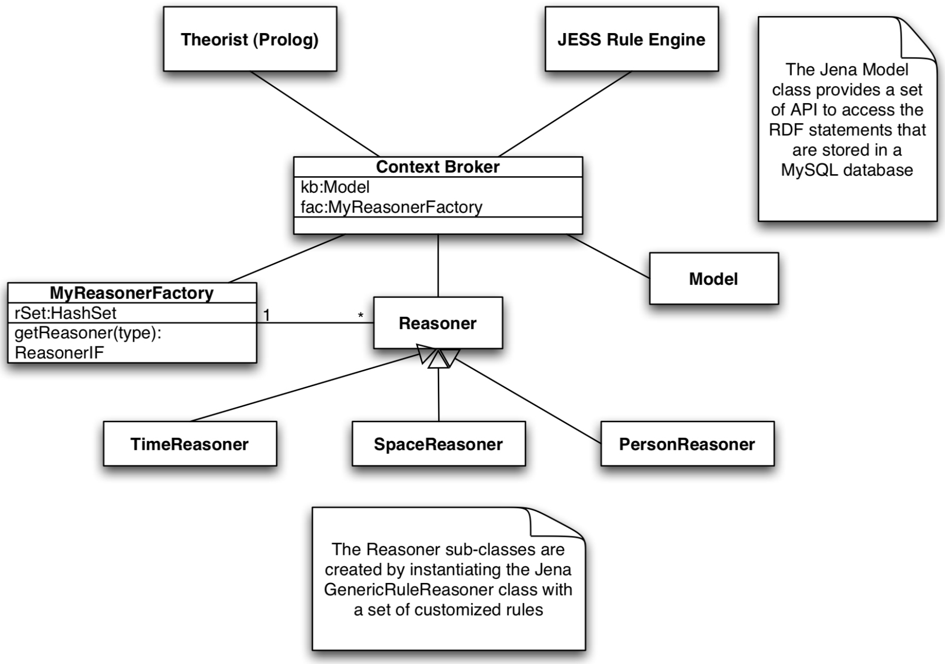

Theorist (Prolog)
JESS Rule Engine
The Jena Model class provides a set of API to access the RDF statements that are stored in a MySQL database
Context Broker
kb:Model
fac:MyReasonerFactory
Model
MyReasonerFactory
rSet:HashSet
getReasoner(type): ReasonerIF
1
*
Reasoner
TimeReasoner
SpaceReasoner
PersonReasoner
The Reasoner sub-classes are created by instantiating the Jena GenericRuleReasoner class with a set of customized rules

Figure 6: A UML state diagram that shows the design of the prototyped context broker implementation. The implementation exploits a number of different rule-based systems to support context reasoning.

necessary logical inferences to resolve inconsistent knowledge. Not all context reasoners are implemented in Java. The context broker implementation uses different programming interfaces to interact with the different rule-based systems. It directly interacts with both JESS and Jena rule engines using their built-in Java API. Because Theorist is a Prolog program, it is executed as a separate running process using the SICStus Prolog's Java Native Interface API.

C. How to Interpret Context

Interpreting context typically involves a two-step process in a context broker. The first step is for the context broker to determine the type of the information that has been acquired and decide whether it can be fused with the existing knowledge to deduce new contextual knowledge. If the new information can be used to infer new contextual knowledge, the second step is for the context broker to load relevant facts and rules into appropriate reasoners and invoke the associated reasoning rules.

I prototyped two kinds of context reasoner implementations. One kind exploits the ontology semantics defined by the OWL language, and the other kind exploits heuristic rules associated with certain application domains. These reasoners can interpret contextual information that is associated with time, space, and meeting events.

1. Temporal Reasoning

Time is a central notion in describing events. In a smart meeting room environment, events may include the occurrence meetings, the presence of people, the performance of actions, the sending and receiving of messages among the agents, and the internal state changes of services and devices. When events are properly described with temporal information, the context broker will be able to reason about their associated temporal order relations.

In the prototype, the logical inference for temporal reasoning is implemented based the semantics of the SOUPA time ontology. The notion of time instant and time interval are the basic primitives in this implementation, and they are used to describe smart meeting room events. The intended usage of this logical inference framework is to compute the temporal ordering of a set of events.

Temporal ordering information is useful to a context broker in a number of different ways. First, it can help a context broker to answer queries with temporal constraints. For example, an agent may wish to know the arrival time of different people who were present in a particular room from 8:00 AM to 12:00 PM. Second, temporal ordering information can help a context broker to correlate distinctive contextual information. For example, knowing a meeting is scheduled to take place from 10:00 AM to 11:00 AM in Room 232, and a cellphone device has been detected in the same room at 10:32 AM, without any evidence to the contrary, a context broker can conclude the owner of the cellphone is attending the meeting. Third, temporal ordering information also can be used to detect inconsistent contextual knowledge. For example, a context broker is informed that a person is currently located at home. Later, a new report indicates that the same person is present at work. The event time intervals described in the two reports overlap. Using this temporal ordering information, the context broker can detect the person's location information is inconsistent.

I implemented two different temporal reasoners. One is in Prolog, and the other is in Jena. The Prolog implementation is intended to provide temporal reasoning support for Prolog applications (e.g., Theorist). The Jena implementation is intended to provide similar support for Java programs that use the Jena API. When processing the temporal descriptions of different events, the reasoners assume time values (i.e., date, time, and time zone) are expressed according to the ISO 8601 Date and Time Formats (see [12] for details). When reason with time values that are expressed in different time zones, the reasoners first computes their corresponding values in UTC (Coordinated Universal Time), and then computes their temporal ordering relation using the UTC referenced values.

The Jena implementation of the reasoner is built on a general purpose rule-based reasoner called Generic Rule Reasoner, which is provided by the Jena API. This reasoner supports rule-based inference over RDF graphs and provides forward chaining, backward chaining, and a hybrid execution model [92]. When using the Jena API to process OWL data, all RDF statements are stored in an abstract data structure called *model*. The model of the RDF statements can answer queries about the underlying RDF graph. A Generic Rule Reasoner can be attached to a model to provide additional inference support. Figure 7 shows some of the rules used to define a temporal reasoner in Jena.

```
(?x tme:before ?y) <-
     (?x rdf:type tme:IntervalThing), (?x tme:ends ?endsX),
     (?y rdf:type tme:IntervalThing), (?y tme:begins ?beginsY),
     (?endsX tme:before ?beginsY).

(?x tme:inside ?y) <-
    (?x rdf:type tme:InstantThing),
    (?y rdf:type tme:IntervalThing),
    (?y tme:begins ?beginsY), (?y tme:ends ?endsY),
    (?beginsY tme:before ?x), (?x tme:before ?endsY).

(?x tme:overlaps ?y) <-
    (?x rdf:type tme:ProperIntervalThing),
    (?y rdf:type tme:ProperIntervalThing),
    (?x tme:begins ?beginsX), (?x tme:ends ?endsX),
    (?y tme:begins ?beginsY), (?y tme:ends ?endsY),
    (?beginsY tme:before ?endsX),
    (?beginsX tme:before ?beginsY),
    (?endsX tme:before ?endsY).

(?x tme:overlappedBy ?y) <- (?y tme:overlaps ?x).
```

Figure 7: An example of the rules used to configure a Generic Rule Reasoner to support temporal reasoning. These rules are defined to reason over the SOUPA time ontology vocabularies in a RDF graph.

2. Spatial Reasoning

Spatial information is an important part of the location context. The logical inference for reasoning over the spatial information of some object's location context is called spatial reasoning. In CoBrA, spatial reasoning is more than just knowing the location of a physical object. It includes the use of geo-spatial data and ontology to derive new spatial knowledge that cannot be obtained through sensing.

In general, the location context of a physical object has two kinds of spatial information: geo-metric and symbolic. Different logical inference for geo-metirc and symoblic information can provide different spatial reasoning support in a context broker. For example, the geo-metric information such as the latitude and longitude coordinates can be used to calculate the distance between two locations. The symbolic representation of things typically found in maps or construction blueprints can be used to support qualitative spatial reasoning

[90].

The SOUPA space ontology is central to the spatial reasoning implementation in CoBrA. The implementation assumes that all spatial information is expressed using the SOUPA space ontologies. Based on the ontology semantics, the following logical inference is supported: (i) given geographical space X and Y (i.e., they are instants of the `spc:GeographicalSpace` class), infer whether X is spatially subsumed by Y or X spatially subsumes Y, (ii) given two geographical space X and Y, infer whether they are disconnected, (iii) given geographical space X with defined latitude and longitude coordinates, infer geographical space Y that spatially subsumes X, and (iv) given geographical space X with defined latitude and longitude coordinates, infer geographical space Y that is spatially subsumed by X.

A context broker can exploit spatial reasoning in a number of different ways. First, it can use spatial reasoning to infer a location's spatial properties that cannot be directly acquired from the physical sensors. For example, from the readings of a GPS device that a person carries, the context broker can acquire some basic information about the person's location (e.g. the latitude and longitude coordinates). By finding the corresponding symbolic representation of the coordinates in a pre-defined ontology, the context broker can infer additional spatial information (e.g., the city in which the person is located, or a nearby building that is spatially subsumed by the university campus that the person is located in). Second, spatial reasoning can used to detect inconsistent information about a person's location. For example, according a predefined ontology, room A is disconnected from room B, and there are two different reports about the same person is located in both room A and room B during the same time interval. Using spatial reasoning, the context broker infers a person cannot be simultaneously located in two disconnected rooms. Hence, it concludes the reported location information is inconsistent. Third, the context broker can use spatial reasoning to protect users' privacy by adjusting the granularity of their location information. For example, based on a set of predefined obfuscation rules, the context broker intentionally generalizes the location information of a person and shares which with the agents that the person does not trust.

I implemented a spatial reasoner using the Jena API's Generic Rule Reasoner. This spatial reasoner is capable of reasoning over the symbolic spatial constructs defined in the SOUPA space and RCC ontology. Figure 8 shows some of the rules used to define a spatial reasoner.

```
(?x spc:spatiallySubsumes ?y) <-
  (?x rdf:type spc:GeographicalSpace), (?y rdf:type spc:GeographicalSpace),
  (?y spc:spatiallySubsumedBy ?x).

(?x spc:spatiallySubsumedBy ?z) <-
  (?x rdf:type spc:GeographicalSpace), (?y rdf:type spc:GeographicalSpace),
  (?x spc:spatiallySubsumedBy ?y), (?y spc:spatiallySubsumedBy ?z).

(?x rcc:isDisconnectedFrom ?y) <-
  (?x rdf:type spc:GeographicalSpace), (?y rdf:type spc:GeographicalSpace),
  (?x rdf:type rcc:RCCSpatialRegion), (?y rdf:type rcc:RCCSpatialRegion),
  (?x spc:spatiallySubsumedBy ?z), (?z rcc:isDisconnectedFrom ?y).

(?x rcc:properPartOf ?y) <-
  (?x rdf:type spc:GeographicalSpace), (?y rdf:type spc:GeographicalSpace),
  (?x rdf:type rcc:RCCSpatialRegion), (?y rdf:type rcc:RCCSpatialRegion),
  (?x spc:spatiallySubsumes ?y), (?y rcc:notPartOf ?x).
```

Figure 8: An example of the rules used to configure a Generic Rule Reasoner to support spatial reasoning. These rules are defined to reason over the SOUPA space and RCC ontology vocabularies in a RDF graph.

3. Meeting Event Reasoning

Part of CoBrA's feasibility demonstration is to support context-aware services in a smart meeting room environment. The role of a context broker in this system to maintain a shared model of the meeting event context and share this information with other services. In the prototype implementation, the context of a meeting event is defined as information that describes (i) the schedule of the meeting and its expected participants, (ii) the situational roles of the individual participants, (iii) the types of services should be provided to the participants based on their situational roles, and (iv) the profiles and privacy policies of the participants.

The context broker uses different kinds of logical inference to interpret meeting event context. This includes the use of temporal and spatial reasoners as well as the use of logical inference rules associated with the application domain heuristics. In the prototype, the domain-specific inference rules are defined based on the conventions of the eBiquity Research Group at UMBC. These conventions are as follows.

- The schedule of the eBiquity group meetings are usually accessible on the group's web site. The web site publishes detail descriptions of the individual meetings in OWL. The ontology used by the web site can be mapped to the SOUPA and CoBrA ontologies.

- Members of the eBiquity group publish their personal profiles and social network information on the Web. The same ontology and instant data used to describe their individual identities are also used by the eBiquity group's web site in publishing meeting attendee descriptions.

- If a visitor from a different organization is scheduled to participate in an eBiquity group meeting, his/her personal profiles and social network information are published on the eBiquity group web site prior to his/her arrival.

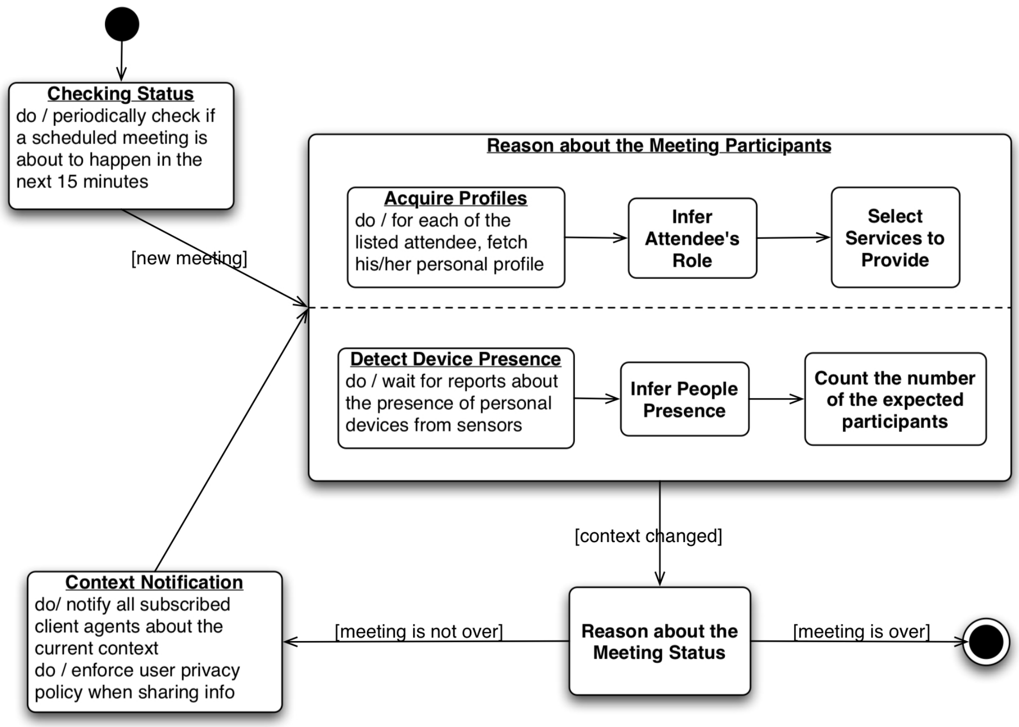

Checking Status
do / periodically check if a scheduled meeting is about to happen in the next 15 minutes
[new meeting]
Reason about the Meeting Participants
Acquire Profiles
do / for each of the listed attendee, fetch his/her personal profile
Infer Attendee's Role
Select Services to Provide
Detect Device Presence
do / wait for reports about the presence of personal devices from sensors
Infer People Presence
Count the number of the expected participants
[context changed]
Context Notification
do/ notify all subscribed client agents about the current context
do / enforce user privacy policy when sharing info
[meeting is not over]
Reason about the Meeting Status
[meeting is over]

Figure 9: A UML diagram that depicts the state transition of a context broker's meeting context reasoning.

- People usually carry some kind of personal devices (e.g., cellphones or PDA) in a smart meeting room environment. The device profile of these personal devices are published on the personal homepages of their users in OWL using the SOUPA device ontology.
- People who are scheduled to give presentations at the eBiquity group meetings typically make available their PowerPoint presentation slides on the Web. The necessary information that are needed to download these documents are described in their personal profiles.

Meeting context reasoning is a continuous process. The implementation of this process can be modeled as a set of state transitions (Figure 9). This process begins as part of a context broker's behavior execution. The first state (Checking Status) involves the reasoning to determine whether or not any scheduled meeting is about to take place in the room. This stage of reasoning uses the meeting schedule information that is acquired by the context broker from the eBiquity group's web site. If no meeting is expected in the next 15 minutes, the Checking Status state remains active. Otherwise, the process enters a new state – "Reasoning About the Meeting Participants".

In this state, there are two parallel processes. One process is responsible for inferring services that should be provided to the meeting participants based on their situational roles. The role information is inferred from the acquired personal profiles. The service selection reasoning is based on a pre-defined user model (i.e., presentation services should be provided to all speakers, greeting services should be provided to all meeting participants). The other process, which is executed in parallel to the previous one, is responsible for counting the number of the arrived participants. This calculation uses the attendee information described in the meeting schedule and the people presence information inferred from the sensing data. Knowing who is expected to participate in the meeting, the reasoner matches this information against the identity of the people who are present in the room. During these two processes, if new knowledge is inferred (i.e., an indication of context change), a new state is entered.

In this state, the process is to reason about the meeting status. The logic of the reasoning is as follows: (i) a scheduled meeting is in session if the current time instant is inside the beginning and the ending time of the meeting schedule, (ii) a scheduled meeting has ended if the current time instant is after the ending time of the meeting schedule, (iii) a participant (a presenter or an invited participant) is arrived at the meeting if he/she is located in the room of the schedule meeting, and there is no evidence suggesting that he/she is located in any other place, and (iv) a meeting can begin if no participants (a presenter or an invited participant) are missing. Figure 10 is the set of JESS rules that implement this logic inference process.

If the meeting status reasoning concludes a scheduled meeting is currently taking place, the process enters a new state. In this new state, the context broker has the opportunity to share its new contextual knowledge with other agents. During the knowledge sharing process, the context broker is required to enforce privacy policies that are defined by the users. On the other hand, if the meeting status reasoning concludes an ongoing meeting is over, then the present reasoning process for this particular meeting is terminated. Unless the whole behavior execution of the context broker is terminated, the meeting event reasoning process continues.

D. How to Detect and Resolve Inconsistency

Different information acquired from distinct resources may be inconsistent with each other. In order to maintain a consistent model of context, a context broker must be able to detect the existence of inconsistency in the knowledge base and resolve it through reasoning and communication.

A number of different reasons could cause information inconsistency. For example, the raw data collected from the physical sensors is inaccurate, the logical inference used to interpret the sensing data has flawed

```
(defrule meeting-in-session
   (meeting (name ?x) (location ?l) (begin-time ?bTime) (end-time ?eTime))
   (current-time ?cTime)
   (test (call XSDDateTimeTool after ?cTime ?bTime))
   (test (call XSDDateTimeTool before ?cTime ?eTime))
   =>
   (assert (meeting-in-session ?x)))

(defrule meeting-has-ended
   ?id <- (meeting-in-session ?x)
   (current-time ?cTime)
   (meeting (name ?x) (end-time ?eTime))
   (test (call XSDDateTimeTool after ?cTime ?eTime))
   =>
   (retract ?id))

(defrule participant-arrived-at-meeting
   (meeting-in-session ?m)
   (meeting (name ?m) (location ?loc) (begin-time ?bTime) (end-time ?eTime))
   (arrival (person ?p) (location ?loc) (at-time ?t))
   (test (call XSDDateTimeTool after ?t ?bTime))
   (test (call XSDDateTimeTool before ?t ?eTime))
   (not (not-located-in ?p ?loc))
   =>
   (assert (arrived ?p ?m)))


(deffunction can-meeting-begin (?meeting)
   (bind ?a1 (count-query-results find-meeting ?meeting))
   (bind ?a2 (count-query-results find-missing-participant ?meeting))
   (if (= ?a1 0) then (return FALSE)
    else (if (= ?a2 0) then (return TRUE)
          else (return FALSE)
         )
   )
)
```

Figure 10: JESS rules that implement the logical inference for reasoning about the meeting status. In order to support the reasoning in JESS, an external procedure is used to map facts from the OWL representation of the meeting schedule to the JESS rule representation.

domain heuristics, or the defined knowledge representation is unable to model every aspects of the physical context in perfection. In CoBrA, the context reasoning attempts to address this problem.

I developed an ontology-based approach for detecting information inconsistency and an assumption-based reasoning approach for resolving information inconsistency. I prototyped implementations to show the feasibility of these approaches.

My implementations support two different scenarios. In one scenario I showed how to detect and resolve the inconsistent information about a person's location, and in the other I showed how to detect and resolve the inconsistent information about the status of a scheduled meeting.

1. Detect Inconsistency Based on the Ontologies

Ontologies expressed using the OWL langauge can be used to detect information inconsistency. The idea is to model context as ontologies and exploit ontology reasoning to detect information inconsistency. The OWL langauge is an expressive language that allows various aspect of a context to be represented. This includes objects in a context domain, relations between these objects, and restrictions of those relations. Once an ontology is defined, context observations made in the physical world are represented as instant data of the ontology. Given an ontology and a set of instant data, ontology reasoners are used to infer whether the context described by the instant data is consistent with the model defined by the ontology.

I developed a prototype implementation to show how this approach is used to detect inconsistent information about a person's location. The application scenario is as follows. Harry enters his a conference room (ITE RM 201). The room detects the presence of his Bluetooth cellphone. Without knowing any evidence to the contrary, the room infers Harry is also present in the room and informs the context broker. Moments later, Harry leaves the conference room in a hurry and forgets his cellphone in the room. After leaving the conference room, Harry goes back to his office (ITE RM 338). As he logs in to his office's workstation, the computer detects his presence and informs the context broker that Harry is located in his office. According to the ontology (see Figure 11), ITE RM 338 is spatially disconnected from ITE RM 201. Given this information and unable to prove Harry has exited from any one of these rooms, the context broker concludes that the location information about Harry is inconsistent.

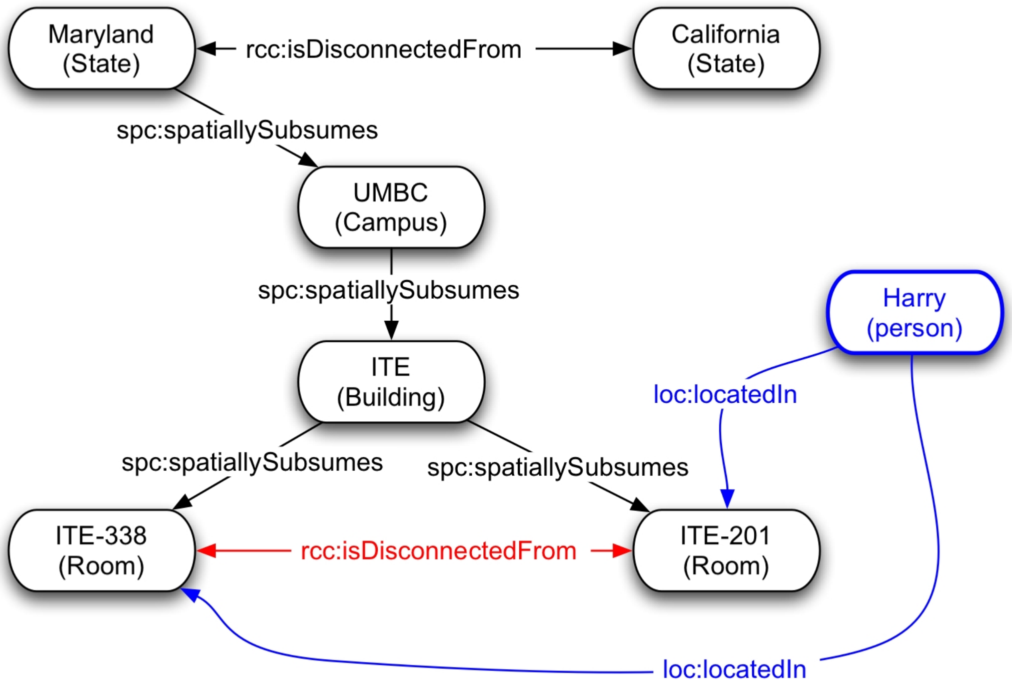


Figure 11: Using ontologies to detect inconsistent location information about a person. Based on the ontology, the context broker infers it is inconsistent for Harry being located in both Room ITE-201 and Room ITE-338.

2. Resolve Inconsistency by Weighing Assumptions

The root of information inconsistency is the source agents that provide the information. Because the agents may have used inaccurate sensing data or flawed domain heuristics to interpret context, information acquired from these agents may be inconsistent. The scenario described in the previous section is a typical example that shows how inconsistent information can emerge due to sensing limitations and flawed reasoning in the source agents.

I developed an assumption-based reasoning approach for resolving inconsistent information. In this approach, the context broker weighs the assumptions that the individual agents used to derive their conclusions and only accept conclusions that have the most reliable assumptions.

Assumptions are the default beliefs of an agent, and they are the foundation of the agent's logical inference. For example, when an agent detects the presence of a cellphone, the agent infers the presence of the owner of the cellphone. In this case, the assumption of the agent is the co-location relation between the cellphone and the person, i.e., the owner of a cellphone is always located in the same place as the cellphone.

Note that the assumptions built into the reasoning of an agent may not always lead to accurate conclusions. These assumptions may have flawed heuristics. For example, an exception in assuming the co-location relation is a situation in which the owner accidently loses or walks away from the cellphone. In such case, the assumption is no longer reliable. Consequently, the conclusion derived from this assumption also becomes unreliable.

I developed two prototype implementations to show the feasibility of this approach. Both implementations use the Theorist reasoning system for modeling and reasoning about assumptions. The difference between the two implementations is in the modeling of the assumptions. In one implementation, the context broker has an explicit model of the assumptions that the other agents use to interpret context. In the other implementation, the context broker does not begin with an explicitly model other agents' assumptions. As the context broker detects inconsistent information, it acquires the assumptions dynamically from the agents via communications.

In my prototype implementations, the reasoning system designs assume the following:

- All ontologies used by the context broker are assumed to be consistent. The context broker is not required to detect and resolve inconsistent information that is resulted from inaccurate ontology specifications.
- All agents share a common ontology with the context broker. This common ontology includes the vocabularies and the associated semantics used for describing contexts and modeling reasoning assumptions.
- When agents are asked by the context broker to justify their beliefs about the context, no agents shall intentionally provide false information to deceive the context broker. In other words, there is no "liar" agents, and all agents are cooperative.
- The context broker is only responsible for resolving the kinds of inconsistent information that it has sufficient knowledge about. The built-in logical inference of the context broker pre-defines this knowledge.

Theorist Framework Overview

For modeling and reasoning about the assumptions of different agents, I used the Theorist system. It is a logical reasoning system for default reasoning and diagnosis [87]. The version of the Theorist that I have used is in Prolog (see [86] for the users guide of this implementation). I adopted the following introduction from the Theorist Users Guide.

When using Theorist, the user provides three sets of first order formulae.

$\mathcal{F}$ is a set of closed formulae called the *facts*. These are intended to be true in the world being modeled.

Δ is a set of formulae which act as *possible hypotheses*, any ground instance of which can be used in an explanation if consistent.

$\mathcal{C}$ is a set of closed formulae taken as constraints. The constraints restrict what can be hypothesized.

We assume that $\mathcal{F} \cup C$ is consistent.

Definition 4 a **scenario** of $\mathcal{F}, \Delta$ is a set $D \cup \mathcal{F}$ where D is a set of ground instances of elements of Δ such that $D \cup \mathcal{F} \cup C$ is consistent.

Definition 5 If g is a closed formula then an **explanation** of g from $\mathcal{F}, \Delta$ is a scenario of $\mathcal{F}, \Delta$ which implies g.

That is, g is explainable from $\mathcal{F}, \Delta$ if there is a set D of ground instances of elements of Δ such that

$\mathcal{F} \cup D \models g$ and

$\mathcal{F} \cup D \cup C$ is consistent

$\mathcal{F} \cup D$ is an explanation of g.

The syntax of Theorist is designed for maximum flexibility. Virtually any syntax is appropriate for wffs; the formulae are translated into Prolog clauses without mapping out subterms.

A **wff** is a well formed formula made up of arbitrary combination of implication (“$=>$”, “$< -$”), disjunction (“or”, “;”), conjunction (“and”, “&”, “,”) and negation (“not”, “~”) of atomic symbols. Variables follow the Prolog convention of being in upper case. There is no explicit quantification.

A **name** is an atomic symbol with only free variables as arguments.

The following gives the syntax of the Theorist code:

fact w.

where w is a wff, means that $(\forall w) \in \mathcal{F}$; i.e., the universal closure of w (all variables universally quantified) is a fact.

default d.

where d is a name, means that $d \in \Delta$; i.e., d is a default (a possible hypothesis).

default $d : w$.

where d is a name and w is a wff means w, with name d can be used in a scenario if it is consistent. Formally it means $d \in \Delta$ and $(\forall d \Rightarrow w) \in \mathcal{F}$.

constraint w.

where w is a wff means $\forall w \in \mathcal{C}$.

prolog p.

where p is an atomic symbol means any Theorist call to p should be proven in Prolog. This allows us to use built-in predicates of pure Prolog. One should not expect Prolog's control predicates to work.

explain w.

where w is an arbitrary wff, gives all explanations of $\exists w$.

predict w.

where w is a arbitrary ground wff, returns "yes" if w is in every extension of the defaults and "no" otherwise. If it returns "yes", a set of explanations is returned, if it returns "no" then a scenario from which g cannot be explained is returned (this follows the framework of [85]).

The following example is a file to do the birdsfly example

```
dyn flies(X).
dyn bird(X).
default birdsfly(X): flies(X) <- bird(X).
constraint not birdsfly(X) <- not flies(X).
default emusdontfly(X): not flies(X) <- emu(X) .
constraint not emusdontfly(X) <- flies(X).
constraint not birdsfly(X) <- emu(X).
fact bird(X) <- emu(X).
fact bird(X) <- robin(X).
fact bird(tweety).
fact emu(polly).
fact robin(cohen).
```

The following is a session with theorist for default reasoning. We assume that the preceding code is in a file called "birdsfly".

```
 theorist

Welcome to THEORIST 1.1 (5 October 88 version)
Any Problems see David Poole (poole@vision.ubc.cdn)

yes
| ?- thconsult birdsfly.

yes
| ?- explain flies(tweety).
Answer is flies(tweety)
Theory is [birdsfly(tweety)]


no
| ?- explain flies(polly).
```

```
no
| ?- explain flies(X).
Answer is flies(cohen)
Theory is [birdsfly(cohen)]

Answer is flies(tweety)
Theory is [birdsfly(tweety)]


no
| ?-
```

Note that the explain command always finds all answers and then returns with “no” (note that, for simplicity, it is a Prolog command).

A Use Case Implementation

When a context broker detects inconsistency, it first tries to find the assumptions that different agents have used to derive their interpretation of the context, and then tries to resolve inconsistent information by adopting the context interpretation with the most reliable assumptions. In one of the prototype implementation scenario, the assumptions of different sensing agents are explicitly represented as part of the context broker’s reasoning framework. The following is the representation of these assumptions as Theorist formulae:

```
fact can_bel(observed(S,located_in(P,R))) <-
    voice_reg_sensor(S), acquired_reliable_voice_data(S,R).

fact can_bel(observed(S,located_in(P,R))) <-
    cellphone_sensor(S), cellphone(C), carries_cellphone(P,C).

default acquired_reliable_voice_data(S,R).

constraint not acquired_reliable_voice_data(S,R) <-
    sensor(S), observed(S,noise_level(R,L)), gt(L,60).

default owner_often_carries_cellphone(P,C) :
   carries_cellphone(P,C) <- person(P), cellphone(C), owned_by(C,P).

constraint not carries_cellphone(P,C) <- lost(P,C).
constraint not carries_cellphone(P,C) <- not_colocate(P,C).

fact not_colocate(P,C) <- person(P), cellphone(C), owned_by(C,P),
    sensor(S),
    observed(S,missed_calls(C,N1)), gt(N1,10),
    observed(S,outgoing_calls(C,N2)), eq(N2,0).

fact voice_reg_sensor(‘http://cobra.umbc.edu/sensor/s1’).
fact cellphone_sensor(‘http://cobra.umbc.edu/sensor/s2’).

fact person(‘http://umbc.edu/˜hchen4’).
fact cellphone(‘http://umbc.edu/˜hchen4/hc_t68i’).
fact owned_by(‘http://umbc.edu/˜hchen4/hc_t68i’,
              ‘http://umbc.edu/˜hchen4’).
```

The term `located_in(P,R)` represents the fact that person P is currently located in room R. The term `observed(S,located_in(P,R))` represents the fact that sensor S has observed person P is located in room R. The term `can_bel(observed(S,located_in(P,R)))` represents the fact that the context broker believes the assumption that sensor agent S uses to interpret its observation about the location of person P. For simplicity, temporal property is not explicitly modeled in this reasoning program.

In this program, the sensor agent's assumptions are defined by two different Theorist defaults: `acquired_reliable_voice_data(S,R)` and `owner_often_carries_cellphone(P,C)`. The first default is part of a voice recognition agent's assumption. The context broker believes that if a voice recognition agent observed a person is located in room R, then the agent must have assumed the voice data collected from room R is reliable. The second default is part of a cellphone presence agent's assumption. The context broker believes that if a cellphone-detection agent observed person P is located in a room, then the agent must have assumed person P always carries a cellphone.

For each default assumption, a set of constraints are defined for weighing the reliability of the conclusion that it derives. In the code above, the context broker is restricted not to hypothesize `acquired_reliable_voice_data(S,R)` if the voice data is collected when the noise level in the room is greater than 60 decibels. The context broker is also restricted not to hypothesize `owner_often_carries_cellphone(P,C)` if either person P has lost his/her cellphone C or person P is not co-located with cellphone C. Person P is not co-located with cellphone C if cellphone C has 10 missed calls and 0 outgoing calls.

Note that for demonstration purposes, these defined constraints only consider a small set of attributes and heuristics, and it represents one possible model for weighing the reliability of other agents' assumptions. Depending on the actual requirements of future applications, these constraint definitions can be fine tuned and extended with additional attributes and heuristics.

When the context broker receives information from the sensor agents, it asserts this information into the Theorist reasoner as a set of facts. For example, when a voice recognition agent reports person Harry Chen is located in Room 338, then context broker asserts the following term:

```
fact observed('http://cobra.umbc.edu/sensor/s1',
        located_in('http://umbc.edu/~hchen4',
  'http://cobra.umbc.edu/ont/2004/05/ebiquity-geo#ITE338')).
```

In this term, the URI's represent the identities of the corresponding sensor agent, person, and room. These URI's are defined as part of the shared ontology between the context broker and the sensor agents.

The next step in the reasoning is to detect if the new information is consistent with the current belief of the context broker. In this step, the detection of inconsistency is done by the ontology reasoning described in the previous section. If inconsistency is detected, then the context broker tries to determine the assumptions used by different sensor agents that provide this information. For example, after being told by the voice recognition agent that Harry Chen is located in Room 338, the context broker receives information from a cellphone-presence detection agent saying that Harry is located at home (which is identified by the URI `http://cobra.umbc.edu/ont/2004/05/ebiquity-geo#LOC232`).

```
fact observed('http://cobra.umbc.edu/sensor/s2',
        located_in('http://umbc.edu/~hchen4',
  'http://cobra.umbc.edu/ont/2004/05/ebiquity-geo#LOC232')).
```

Knowing Room 338 is spatially disconnected from Harry's home, the context broker concludes there is an inconsistency about Harry's location.

Using the Theorist system, this inconsistency can also be detected by reasoning if the context broker's knowledge about Harry's location (represented by a Theorist formulae ω), is in every extension of the defaults. In specific, based on the facts that are told by different sensor agents, the context broker may have a set of knowledge K that describe Harry's location (i.e., $K = \{\omega_1 \ldots \omega_n\}$). In the previous example, $K = \{\omega_1, \omega_2\}$, and ω_1 represents Harry is located in Room 338, and ω_2 represents Harry is located at home. To detect inconsistency, the context broker reasons for each ω_i in K, if it is in every extension of the defaults. Based on the code described in above, we augment the reasoning with an additional set of formulae:

```
default obs_person_at_work(S,P,R):
  located_at_work(P) <- sensor(S), conf_room(R),
     observed(S,located_in(P,R)),
     can_bel(observed(S,located_in(P,R))).

default obs_person_at_home(S,P,R):
  located_at_home(P) <- sensor(S), room_at_home(R),
    observed(S,located_in(P,R)),
    can_bel(observed(S,located_in(P,R))).

fact not (located_at_home(P) and located_at_work(P)).

fact pred_person_loc(P,work) <- located_at_work(P).
fact pred_person_loc(P,home) <- located_at_home(P).

fact room_at_home(R) => room(R).
fact conf_room(R) => room(R).

fact voice_reg_sensor(S) => sensor(S).
fact cellphone_sensor(S) => sensor(S).

fact conf_room('http://cobra.umbc.edu/ont/2004/05/ebiquity-geo#ITE338').
fact room_at_home('http://cobra.umbc.edu/ont/2004/05/ebiquity-geo#LOC232').
```

The named default "`obs_person_at_work`" represents the fact that person P is located at work if a sensor has observed person P is located in a conference room, and the context broker believes the assumption that the sensor has used. The named default "`obs_person_at_home`" has similar logic. It represents the fact that person P is located at home if a sensor has observed person P is located in a room that is part of the home, and the context broker believes the assumption that the sensor has used. These two named defaults are possible hypothesis for explaining the location of a person, which is represented by the predicate "`pred_person_loc`".

It is inconsistent to believe a person is simultaneously located at home and at work. This fact is represented by

```
fact not (located_at_home(P) and located_at_work(P)).
```

To reason if the knowledge about a person's location in every extension of the defaults, the context broker calls the "**predict**" command of the Theorist system. The following is a command line session that shows how the reasoner detects inconsistency because "`pred_person_loc(P,R)`" is not in every extension of the defaults:

```
SICStus 3.11.2 (x86-win32-nt-4): Wed Jun  2 19:48:46 WEDT 2004
Licensed to umbc.edu
| ?- thconsult demoloc2.
yes
| ?- fact observed('http://cobra.umbc.edu/sensor/s1',
  located_in('http://umbc.edu/~hchen4',
  'http://cobra.umbc.edu/ont/2004/05/ebiquity-geo#ITE338')).
yes
| ?- fact observed('http://cobra.umbc.edu/sensor/s2',
  located_in('http://umbc.edu/~hchen4',
  'http://cobra.umbc.edu/ont/2004/05/ebiquity-geo#LOC232')).
yes
| ?- predict pred_person_loc('http://umbc.edu/~hchen4',work).
No, pred_person_loc('http://umbc.edu/~hchen4',work) is not explainable from
[acquired_reliable_voice_data('http://cobra.umbc.edu/sensor/s1',
'http://cobra.umbc.edu/ont/2004/05/ebiquity-geo#ITE338'),
obs_person_at_home('http://cobra.umbc.edu/sensor/s2','http://umbc.edu/~hchen4',
'http://cobra.umbc.edu/ont/2004/05/ebiquity-geo#LOC232'),
owner_often_carries_cellphone('http://umbc.edu/~hchen4',
'http://www.cs.umbc.edu/~hchen4/hc_t68i')].
yes
| ?- predict pred_person_loc('http://umbc.edu/~hchen4',home).
No, pred_person_loc('http://umbc.edu/~hchen4',home) is not explainable from
[owner_often_carries_cellphone('http://umbc.edu/~hchen4',
'http://www.cs.umbc.edu/~hchen4/hc_t68i'),
obs_person_at_work('http://cobra.umbc.edu/sensor/s1',
'http://umbc.edu/~hchen4','http://cobra.umbc.edu/ont/2004/05/ebiquity-geo#ITE338'),
acquired_reliable_voice_data('http://cobra.umbc.edu/sensor/s1',
'http://cobra.umbc.edu/ont/2004/05/ebiquity-geo#ITE338')].
yes
```

After detecting inconsistency, the context broker calls the Theorist "**explain**" command to determine the assumptions of the sensor agents. The following is a command line session of the reasoning:

```
| ?- explain can_bel(observed('http://cobra.umbc.edu/sensor/s2',
  located_in('http://umbc.edu/~hchen4',
  'http://cobra.umbc.edu/ont/2004/05/ebiquity-geo#LOC232')).

Answer is can_bel(observed('http://cobra.umbc.edu/sensor/s2',
located_in('http://umbc.edu/~hchen4',
'http://cobra.umbc.edu/ont/2004/05/ebiquity-geo#LOC232')))
Theory is [owner_often_carries_cellphone('http://umbc.edu/~hchen4',
'http://www.cs.umbc.edu/~hchen4/hc_t68i')]
took 0.000 sec.

| ?- explain can_bel(observed('http://cobra.umbc.edu/sensor/s1',
  located_in('http://umbc.edu/~hchen4',
  'http://cobra.umbc.edu/ont/2004/05/ebiquity-geo#ITE338')).

Answer is can_bel(observed('http://cobra.umbc.edu/sensor/s1',
located_in('http://umbc.edu/~hchen4',
'http://cobra.umbc.edu/ont/2004/05/ebiquity-geo#ITE338')))
Theory is [acquired_reliable_voice_data('http://cobra.umbc.edu/sensor/s1',
'http://cobra.umbc.edu/ont/2004/05/ebiquity-geo#ITE338')]
took 0.000 sec.

no
| ?-
```

The inferred answers are exactly the set of assumptions that the voice recognition agent and the cellphone-presence detection agent used to interpret Harry's location – `acquired_reliable_data` and `owner_often_carries_cellphone`. To weigh these assumptions, the context broker adopts a strategy that compares their reliability based on its own internal measurement functions.

These functions consist of (i) verifying if it is true that the noise level in the room was above a pre-defined threshold when the voice recognition agent acquired the voice data, and (ii) verifying if it is true that there is any evidence showing the cellphone was not in the close vicinity of its owner user when the cellphone-presence detection agent discovered the cellphone. For a given assumption, if an internal measurement function returns true, then the assumption is consider unreliable. Once all assumptions have been measured, the context broker adopts only one that is considered to be reliable.

An example of this process is as follows. To decide if the voice recognition agent's assumption is reliable, the context broker asks for the noise level in the room when it detected Harry's presence. The sensor agent replies the noise level was measured at 50 decibels. To decide if the cellphone-presence detection agent's assumption is reliable, the context broker asks for the number of outgoing calls and the number of missed calls recorded in Harry's cellphone. The sensor agent replies there was 11 missed calls and 0 outgoing calls recorded in Harry's cellphone. The context broker asserts this information into the reasoner, and calls "**predict**" again to check if this inconsistency can be resolved.

Based on the defined assumption model, the voice recognition agent's assumption is reliable because the noise level was measured at less than 60 decibels. However, the cellphone-presence detection agent's assumption is unreliable because the number of missed calls is greater than 10 and the number of outgoing calls was 0, i.e., a strong evidence showing that the cellphone and the owner did not co-located in the same room. The following is a command line session of this reasoning process:

```
| ?- fact observed(‘http://cobra.umbc.edu/sensor/s2’,
      missed_calls(‘http://www.cs.umbc.edu/˜hchen4/hc_t68i’,11)).
yes
| ?- fact observed(‘http://cobra.umbc.edu/sensor/s2’,
      outgoing_calls(‘http://www.cs.umbc.edu/˜hchen4/hc_t68i’,0)).
yes
| ?- fact observed(‘http://cobra.umbc.edu/sensor/s1’,
      noise_level(‘http://cobra.umbc.edu/ont/2004/05/ebiquity-geo#ITE338’,50)).
yes
| ?- predict pred_person_loc(‘http://umbc.edu/˜hchen4’,X).
Yes, pred_person_loc(‘http://umbc.edu/˜hchen4’,work) is in all extensions.
Explanations are:
1: [obs_person_at_work(‘http://cobra.umbc.edu/sensor/s1’,
‘http://umbc.edu/˜hchen4’,‘http://cobra.umbc.edu/ont/2004/05/ebiquity-geo#ITE338’),
acquired_reliable_voice_data(‘http://cobra.umbc.edu/sensor/s1’,
‘http://cobra.umbc.edu/ont/2004/05/ebiquity-geo#ITE338’)].
X = work ? ;
no
| ?-
```

I implemented this use case scenario using the JADE agent library. The context broker calls the Theorist reasoner using the Java Native Interface API provided by the SICStus Prolog system. The context broker communicates with the sensor agents using the FIPA messages. The message content is expressed using the OWL language. A special XML transformation process is called to transform the RDF representation of the

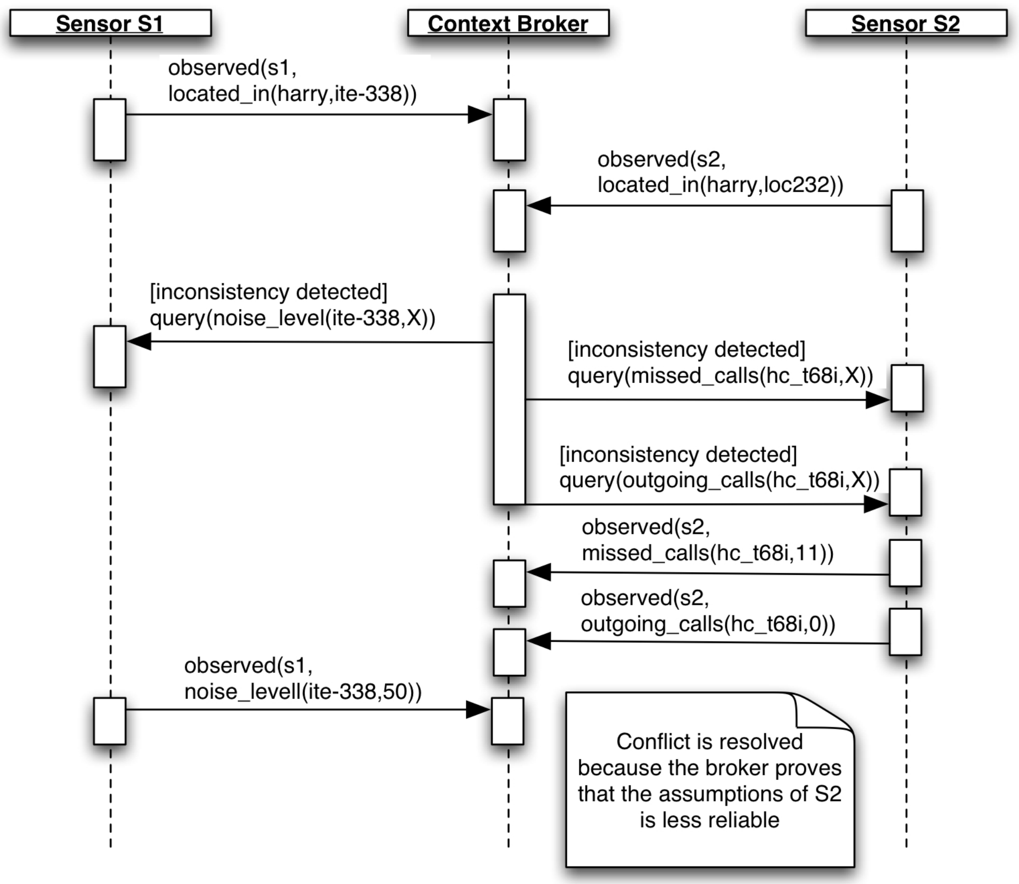

Sensor S1
Context Broker
Sensor S2
observed(s1, located_in(harry,ite-338))
observed(s2, located_in(harry,loc232))
[inconsistency detected] query(noise_level(ite-338,X))
[inconsistency detected] query(missed_calls(hc_t68i,X))
[inconsistency detected] query(outgoing_calls(hc_t68i,X))
observed(s2, missed_calls(hc_t68i,11))
observed(s2, outgoing_calls(hc_t68i,0))
observed(s1, noise_levell(ite-338,50))
Conflict is resolved because the broker proves that the assumptions of S2 is less reliable

Figure 12: This example shows how a context broker can use assumption-based reasoning to resolve inconsistent information. When conflicts are detected, the context broker reasons about the assumptions that different sensors may have used to interpret context. By finding evidences to prove some assumptions are unreliable, the context broker resolves the conflicts between the competing context interpretations.

message content to the corresponding Theorist formulae. Figure 12 shows a UML sequence diagram of this prototype implementation.

3. Reasoning that Incorporates Argumentation

In a dynamic and open environment, a single context broker is usually fragile and has limited capability. To improve system robustness, multiple context brokers may be deployed as a team to support a pervasive context-aware system. In a context broker team, individual context broker is responsible to maintain distinctive contextual knowledge and share this knowledge with other members to create a comprehensive view of the shared context.

Different team members can have different interpretations about the context. Because of this difference, when sharing their knowledge, information inconsistency can occur. To overcome this problem, the context brokers must be able to detect and resolve inconsistency.

This problem can be solved by the assumption-based approach described in the previous section. However, the resulting implementation would require replicating the assumption models of different team members in every single context brokers. A change in the assumption model of a single context broker would directly affect the implementation of other team members.

An alternative approach is to enable the team members to resolve inconsistency through arguments. In this approach, the communications among the team members are defined with a set of special speech acts and protocols. Based on these constructs, the context brokers can be programmed to engage in an argument-based dialog that is aimed for detecting and resolving inconsistent information.

I have developed a prototyped implementation based on the Fatio Argumentation Protocol [73] and simulated the underlying argumentation reasoning of a single context broker in the Theorist system.

In this prototype implementation, the reasoning system design adopts the set of assumptions described in previous section. In addition, it assumes the following:

- When multiple context brokers engage in an argument-based dialog, they share a common ontology that describes the kinds of justification knowledge that can be used in the dialog. In other words, a context broker is only required to provide justifications for knowledge that is part of its built-in logical inference.

- Different context brokers may possess different rules for interpreting the same type of context. However, all context brokers share the same representation of the constructs that are used to express rules.

- In the implementation of each argument-based dialog, there is a special set of knowledge that all context brokers are willing to believe without any doubts. During a dialog, if a context broker asserts some knowledge of this kind is true, the context broker will not be required to justify why this knowledge is true.

A Use Case Scenario

Two context brokers are teamed to support a smart meeting room. Each member of the team has access to different source of contextual information. Periodically the team members synchronize their knowledge through communication. On a typical workday, person Harry is scheduled to host a meeting. Few minutes before the scheduled meeting starts, one member of the context broker team B1 has evidence that Harry is absent from the meeting, and decide that it should inform the meeting schedule to either delay or cancelled the meeting. Before it sends out this notification, it must also obtain consensus from the other team member B2. Context broker B1 informs context broker B2 that Harry is absent from meeting. Based on a different set of sensing information, context broker B2 does not believe Harry is absent. In order to reach a consistent view of the shared context, the context brokers begin an argument-based dialog.

During the argumentation, the objective of context broker B1 is to provide context broker B2 with arguments that can prove the absence of Harry, and the objective of the context broker B2 is to challenge the arguments provided by context broker B2 until it reaches one of the two possible conclusions: it is convinced that the context broker B1's original proposition is either true or false.

The Fatio Argumentation Protocol Overview

In this section, I summarize the key speech acts of the Fatio protocol that are used in my prototype. The following material is adopted from [73].

The statements made in an argumentation dialog may any of: factual statements, expressive statements, social connection statements, commissives, directives, and inferences. Fatio is a generic protocol which can be used on its own, or incorporated into other interaction protocols or ACLs (including FIPA ACL).

The syntax for utterance is:

- *illocution*(P_i, ϕ) or
- *illocution*(P_i, P_j, ϕ)

where *illocution* is an illocution, P_i is an identifier for the agent making the utterance (the speaker), P_j denotes an agent at whom the utterance is directed, and ϕ is the content of the utterance.

Some of the legal locutions in Fatio are the following:

assert(P_i, ϕ): A speaker P_i asserts a statement $\phi \in \mathcal{C}$ (a belief, an intention, a social connection, an external commitment, etc.) In doing so, P_i creates a dialectical obligation within the dialog to provide justification for ϕ if required subsequently by another participant.

question(P_j, P_i, ϕ): A speaker P_j questions a prior utterance of *assert*(P_i, ϕ) by another participant P_i, and seeks a justification for ϕ. The speaker P_j of the question creates no dialectical obligation on himself by the question utterance.

justify(P_i, $\Phi \vdash^{+} \phi$): A speaker P_i who had uttered *assert(P_i, ϕ)*, and was then questioned or challenged by another speaker, is able to provide a justification $\Phi \in \mathcal{A}$ for initial statement ϕ by means of this locution. $\Phi \vdash^{+} \phi$ indicates that Φ is an argument in support of ϕ.

As part the protocol, these locutions are subject to several combination rules:

- The utterance *assert(P_i, ϕ)* may be made at any time.
- The utterance *question(P_j, P_i, ϕ)* may be made at any time following an utterance of *assert(P_i, ϕ)*. Similarly, the utterance *question(P_j, P_i, ϕ)* may be made at any time following an utterance of *justify(P_i, $\Phi \vdash^{+} \phi$)*.
- Immediately following an utterance of *question(P_j, P_i, ϕ)*, the speaker P_i of *assert(P_i, ϕ)* must reply with *justify(P_i, $\Phi \vdash^{+} \phi$)*, for some $\Phi \in \mathcal{A}$.

Context Broker B2's Knowledge in Theorist

I developed a Theorist program to demonstrate the feasibility of using argumentation for resolving inconsistent information. This program simulates the underlying reasoning of context broker B2, which is described in the previous use case scenario. The logical inference of this program is designed to drive the high-level argumentation behavior of the context broker.

Central to this program is a model of context broker B2's belief, which consists of facts that the context broker believes and reasoning rules that it uses in making hypotheses.

```
% D1
default organizer_likely_to_be_present(Person,Meeting):
  organizer_present(Person,Meeting) <-
    meeting(Meeting), organizer(Person, Meeting), at_work(Person).

% D2
default absent_organizer_is_at_home(Person, Meeting) :
  organizer_absent(Person,Meeting) <-
    meeting(Meeting), organizer(Person, Meeting), at_home(Person).
% D3
fact not (organizer_present(Person,Meeting) and
 organizer_absent(Person,Meeting)).
```

In the defined model, the context broker can hypothesize that a meeting organizer is likely to be present at the meeting if it can hypothesize that the organizer is at work. On the other hand, the context broker can hypothesize that a meeting organizer is likely to be absent from the meeting if it can hypothesize that the organizer is at home. In addition, it is inconsistent to hypothesize that the same meeting organizer is likely to be present and absent from the meeting.

```
default at_work_on_workday(Person) :
    at_work(Person) <- person(Person), today(X), is_workday(X).

default at_work_doing_overtime(Person):
    at_work(Person) <- person(Person), today(X), is_weekend(X),
        work_overtime(Person).

constraint not at_work(Person) <- owned_car_at_home(Person).
constraint not at_work(Person) <- on_travel(Person).

default at_home_bec_owned_car_is_at_home(Person):
    at_home(Person) <- person(Person), owned_car_at_home(Person).

default at_home_on_weekend(Person) :
    at_home(Person) <- person(Person), today(X), is_weekend(X).

constraint not at_home(Person) <- work_overtime(Person).
constraint not at_home(Person) <- on_travel(Person).

fact is_workday(X) <-
 (eq(X,mon) or eq(X,tue) or
  eq(X,wed) or eq(X,thur) or eq(X,fri)).

fact is_weekend(X) <-
 (eq(X,sat) or eq(X,sun)).
```

There are two possible defaults for hypothesizing a person is at work: (i) today is a workday, and (ii) today is a weekend and the person working overtime. The context broker cannot hypothesize a person is at work if it believes either an owned car of the person is parked at home, or the person is travelling out of town.

There are two possible defaults for hypothesizing a person is at home: (i) today is a weekend day, and (ii) an owned car of the person is parked at home. The context broker cannot hypothesize a person is at home if it believes either the person is working overtime, or the person is travelling out of town.

Context Broker B2's Argumentation Behavior

In order for agents to engage in an argument-based dialog, they must communicate by following a pre-defined protocol. The communication behavior of context broker B2 is defined based on the Fatio Argumentation Protocol. Algorithm 1 defines this behavior in terms a computational procedure.

This algorithm takes three input arguments: $\mathcal{KB}$ – the context broker's knowledge base, $\mathcal{R}$ – the reasoner built on the Theorist system, and $\mathcal{M}$ – a message queue object for sending and receiving ACL messages.

The algorithm begins with the initialization of the reasoner (line 1). While the communication message queue between the team members is open (line 2), context broker B2 continues to monitor for incoming ACL messages. If the locution of a received message is "assert", then the statement described in the message is evaluated for consistency by calling the utility function CANJUSTIFY.

The CANJUSTIFY function returns true if the statement is consistent with the belief in the knowledge base and returns false otherwise. This function uses the Theorist reasoner to determine if the input statement is consistent.

Algorithm 1 Context broker B2's argumentation behavior

```
B2-ARGUMENTATION-BEHAVIOR($\mathcal{KB}$, $\mathcal{R}$, $\mathcal{M}$)
1: LOAD-KNOWLEDGE($\mathcal{KB}$, $\mathcal{R}$)
2: **while** ISOPEN($\mathcal{M}$) $=$ TRUE **do**
3:     $c \leftarrow$ RECEIVE($\mathcal{M}$, assert)
4:     **if** CANJUSTIFY($\mathcal{M}$, $\mathcal{R}$, $c$) $=$ TRUE **then**
5:         "adopt $c$ because it is consistent"
6:     **else**
7:         **error** "cannot adopt $c$ because it is inconsistent"
8:     **end if**
9: **end while**

CANJUSTIFY($\mathcal{M}$, $\mathcal{R}$, $c$)
1: **if** TYPEOF($c$) $=$ fact **then**
2:     FACT($c$, $\mathcal{R}$)
3: **end if**
4: $result \leftarrow$ PREDICT($c$, $\mathcal{R}$)
5: **if** $result =$ YES **then**
6:     return TRUE
7: **else if** $result =$ NO && TYPEOF($c$) $=$ fact **then**
8:     RETRACT($\mathcal{R}$,$c$)
9:     return FALSE
10: **else if** $result =$ NO && TYPEOF($c$) $=$ hypothesis **then**
11:     SEND($\mathcal{M}$, question($c$))
12:     $z \leftarrow$ RECEIVE($\mathcal{M}$, justify)
13:     return CANJUSTIFY($\mathcal{M}$, $\mathcal{R}$, $z$)
14: **end if**
```

There are two types of statements: fact and hypothesis. Fact statements are propositions that an agent is willing to believe without any justification. Hypothesis statements are propositions that an agent is willing to believe only if they can be justified. In the context of the Theorist system, the fact statements are Theorist "fact" formulaes, and the hypothesis statements are Theorist "default" formulaes.

A key part of the CANJUSTIFY function is to decide when the context broker should adopt a statement that is asserted by the other agent, and when to question the other agent for the justifications of the statement. If the statement is type of "fact", it is temporarily added to the reasoner (line 2). The Theorist "predict" command is called to check if the statement is consistent (line 4). If the "predict" command returns "yes", it implies the current statement c is consistent with respect to the context broker's current belief. The function exits by returning "true" (line 6). On other hand, if the "predict" command return "no", it implies the statement is inconsistent with respect to the context broker's current belief.

If the inconsistent statement is type of "hypothesis", context broker B2 would give an opportunity to the sender agent to argue for a justification of the statement. The context broker sends a question message to the sender agent (line 11). If the sender agent replies with a justification, then the new justification is checked by recursively calling the CANJUSTIFY function (line 12-13). If the inconsistent statement is type of "fact", context broker B2 retracts the statement that was added to the reasoner and exits the function by returning "false" (line 8-9).

A Walk Through

In this section, I describe the dialog between context broker B1 and B2 in an attempt to resolve inconsistent information. This dialog is based on the previously described use case scenario. In this example, the following are facts believed by both context broker B1 and B2:

- Harry is a person, and he is the organizer of meeting M1.
- The day on which the dialog takes place is a Monday, which is a workday.

The following are defaults that are believed by the individual context brokers:

- B1 believes Harry is likely to be absent from meeting M1; B2 believes Harry is likely to be present at meeting M1.
- B1 believes Harry's car is parked at home; B2 has no knowledge about the location of Harry's car.

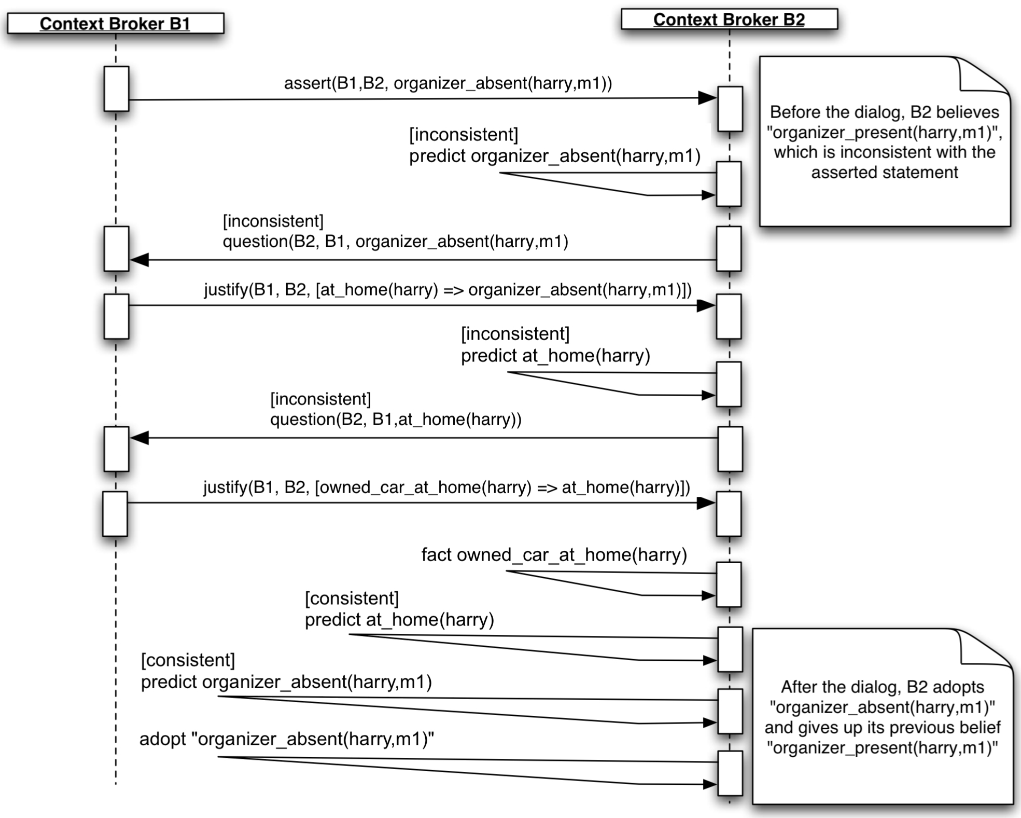

Context Broker B1
Context Broker B2
assert(B1,B2, organizer_absent(harry,m1))
Before the dialog, B2 believes "organizer_present(harry,m1)", which is inconsistent with the asserted statement
[inconsistent]
predict organizer_absent(harry,m1)
[inconsistent]
question(B2, B1, organizer_absent(harry,m1)
justify(B1, B2, [at_home(harry) => organizer_absent(harry,m1)])
[inconsistent]
predict at_home(harry)
[inconsistent]
question(B2, B1,at_home(harry))
justify(B1, B2, [owned_car_at_home(harry) => at_home(harry)])
fact owned_car_at_home(harry)
[consistent]
predict at_home(harry)
[consistent]
predict organizer_absent(harry,m1)
adopt "organizer_absent(harry,m1)"
After the dialog, B2 adopts "organizer_absent(harry,m1)" and gives up its previous belief "organizer_present(harry,m1)"

Figure 13: An argument-based dialog between two context brokers. When context broker B2 detects the statement asserted by context broker B1 is inconsistent, it requests context broker B1 to provide justification. At the end of the dialog, context broker B1 convinces context broker B2 to accept its original assertion.

- B1 believes Harry is at home because his car is parked at home, and B2 believes Harry is at work because today is a workday.

Figure 13 is a UML sequence diagram of the dialog between context broker B1 and B2. The argument begins when context broker B2 detects the statement asserted by context broker B1 is inconsistent with its current beliefs. Through a series exchange of “question-justify” messages, eventually, context broker B2 is convinced that Harry is likely be absent from meeting M1 because his car is parked at home. Figure 14 shows the underlying reasoning of context broker B2 in a Theorist command line session.

```
SICStus 3.11.2 (x86-win32-nt-4): Wed Jun  2 19:48:46 WEDT 2004
Licensed to umbc.edu
| ?- thconsult demomtg1.
yes
| ?- fact today(mon).
yes
| ?- predict organizer_absent(harry,m1).
no
| ?- predict organizer_present(harry,m1).
Yes, organizer_present(harry,m1) is in all extensions.
Explanations are:
1: [organizer_always_present(harry,m1),
at_work_on_workday(harry)].
yes
| ?- predict at_home(harry).
no
| ?- fact owned_car_at_home(harry).
yes
| ?- predict  owned_car_at_home(harry).
Yes, owned_car_at_home(harry) is in all extensions.
Explanations are:
1: [].
yes
| ?- predict at_home(harry).
Yes, at_home(harry) is in all extensions.
Explanations are:
1: [at_home_bec_owned_car_is_at_home(harry)].
yes
| ?- predict organizer_absent(harry,m1).
Yes, organizer_absent(harry,m1) is in all extensions.
Explanations are:
1: [absent_organizer_is_at_home(harry,m1),
at_home_bec_owned_car_is_at_home(harry)].
yes
| ?- predict organizer_present(harry,m1).
no
| ?- halt.
```

Figure 14: A command line session of context broker B2’s argumentation reasoning in the Theorist system.

Chapter VI.

PRIVACY PROTECTION

Privacy is a key issue in the future pervasive computing systems [68, 16, 10, 47]. Ubiquitous sensors will acquire information about the users, and share it with different agents in the environment. In CoBrA, part of a context broker's responsibility is to protect the privacy of the users when sharing their contextual information.

CoBrA adopts a policy-based approach to protect privacy. Policies are defined using the SOUPA policy ontology. Using this ontology, CoBrA allows the users to control the granularity of the information that a context broker can share with other agents. I chose to use the SOUPA policy ontology because it supports policy reasoning using description logic. For this reason, the CoBrA's policy reasoning implementation can exploit the use of an "off-the-shelf" description logic reasoner (i.e., Racer [48]).

However, at the same time, the use of the SOUPA policy ontology brings about tradeoffs in policy expressiveness. In particular, the ontology does not support the definition of variables and rule conditions that are logical expressions (e.g., conjunctions and disjunctions). Without these constructs, for example, users cannot define conditional policy rules or define meta-policies for policy managements [58]. Nevertheless, I believe it is a sensible choice to choose a simple policy language because it allows the re-use of the existing DL-reasoners, which are well-developed and have scalable performance. In the future, when the OWL standards for representing rules becomes available [54], CoBrA's policy representation could be replaced with a new ontology to enhance the expressiveness of privacy policy rules.

A. Background

Policy is an emerging technique for controlling and adjusting low-level system behaviors by specifying high-level rules [100]. The use of policy is common in computing systems that feature security or privacy protection [104]. In typical policy-based systems, policy rules are defined using declarative policy languages that are distinct from the actual system programming languages. A key advantage of using declarative languages to express policies is that the defined policies are more suitable for humans to view and edit. In addition, by separating the logic (i.e., policy rules) from the control (i.e., programming implementations) of the system implementations, policy-based security and privacy protection systems are typically more flexible and adaptable than other non-policy-based systems [100].

Different research works in the policy domain often share a common vision on the use of policy, i.e., using high-level policy rules to control low-level system behaviors. However, they usually adopt different representation languages to define policies. The choice of a representation language can affect the expressiveness and the flexibility of the policy language. For example, the Ponder policy language [35] has a representation that is similar to a typical procedural language, and therefore it is less expressive than other policy languages that are represented using markup languages (e.g., XML and RDF) or Semantic Web languages (e.g., RDFS [14], DAML+OIL [31], and OWL [101]). The P3P (Platform for Privacy Preferences) language [33] has representations in both XML and RDF. Allowing for more expressive constructs, the KAoS policy language [13] is more expressive than the previous two languages. It is defined using the DAML+OIL language. The Rei policy language [58] and the privacy policy language in the e-Wallet system [47] both adopt the OWL language as the representation language for policies.

Policy languages that are represented using the Semantic Web languages are defined in terms of ontologies. Different ontology organizations require different computing approaches to reason and analyze policies. The design of the KAoS policy ontology suggests the use of a description logic inference engine to analyze policy rules. The Rei policy ontology, on the other hand, requires the use of a Frame Logic (F-Logic) based meta-interpreter to compute policy restrictions and constraints. The policy analysis mechanism in the e-Wallet system exploits the XSLT technology to translate policy rules from RDF to JESS rules, and then it use a JESS rule engine to compute policy restrictions.

The SOUPA policy language is similar to those ontology-based policy languages described previously. It also uses a Semantic Web language to represent policies. The SOUPA policy ontology is similar to Rei in modeling a policy as a set of rules that define restrictions on actions (e.g., an action with certain properties is permitted or forbidden). However, it has limited support for meta-policy reasoning (e.g., conflict resolution) and speech acts (e.g., delegation and revocation) [58]. While the SOUPA policy language uses a different representation than the KAoS policy languages, but they share a similar policy reasoning mechanism. They both exploits the use of description logic reasoning.

B. Privacy Protection Issues

Privacy is about the control of information – who has the access to what information. In a pervasive computing environment, people express concerns about their privacy because hidden sensors are embedded in their everyday living environment, and the information acquired by these sensors are often shared by different service agents. Granted that the pervasive computing services will greatly enhance people's computing experiences. However, I believe people will abandon the most useful services if they are unable to take control of their private information.

In CoBrA, the following privacy protection issues are considered: how policy can be used to control the sharing of users' private information, and how users can control the granularity of the information that the system shares.

To protect privacy using policy, an obvious approach is to define a global privacy policy. One policy applies to all users in a space. The context broker of the space enforces this policy whenever it shares users' information. However, this is an inadequate solution. In reality, because different users can have different privacy preferences – e.g., some users may be more comfortable to share their private information with the systems than the others, it would be difficult to capture all privacy preferences of different users in a single global policy. For this reason, the privacy protection of a context broker must allow users to define policies of their own privacy preferences.

Users desire privacy protection because they are concerned about the possible misuse of their information. However, this does not necessary imply that the users desire all of their information to be hidden from the context-aware systems. Completely prohibiting context-aware systems to share their information would hinder the systems' ability to provide relevant services and information. A good example of this situation is the sharing of a user's location information. In a smart meeting room, a context broker constantly tracks a person's location, and context-aware agents often acquire this information from the context broker. For example, while a person is located in a meeting room, she may desire the context broker to share the details of her location only with the agents that she trusts. When some untrusted agent asks for this information, she desires the sharing of information to only reveal general knowledge of her whereabouts.

C. The Policy Ontology

The SOUPA policy ontology is designed based on the following principle: *policies are rules for regulating the permissions for computing entities to perform actions*. In a pervasive computing environment, policies are defined by the human users to permit or forbid computing entities to perform different types of actions. An action can be an invocation to a computing procedure to acquire user information or to access some service in the computing environment. For example, actions can be system calls to low-level API's, remote procedures, or web service interfaces, and actions also can be queries to some persistent data repository or communication messages that are sent to agents in the system.

An action is defined with the following properties:

- **Actor**: an agent that performs the action.
- **Recipient**: an agent that receives the effect after the action is performed.
- **Target**: a physical or an abstract object that the action applies to.
- **Location**: a place at where the action is performed.
- **Time**: time at which the action is performed.
- **Instrument**: a physical or an abstract thing that the actor uses to perform the action.

To regulate the performance of an action, a user can define policy rules to either *permit* or *forbid* the action to be performed. In a pervasive computing system, the enforcement of a policy can be the duty of a pre-established central authority or part of the obligation of a self-governing agent.

Policies are documents. All defined policies possess similar attributes that describe typical written documents (e.g., meta-data about the documents). The set of typical policy document attributes include the author who created the policy, the creation time of the policy, the version information of the policy, and the expected policy enforcer of the policy.

In a policy document, it is often infeasible to define policy rules for every possible actions of a domain. For example, in a location-aware system, different agents may be interested to know the whereabouts of a

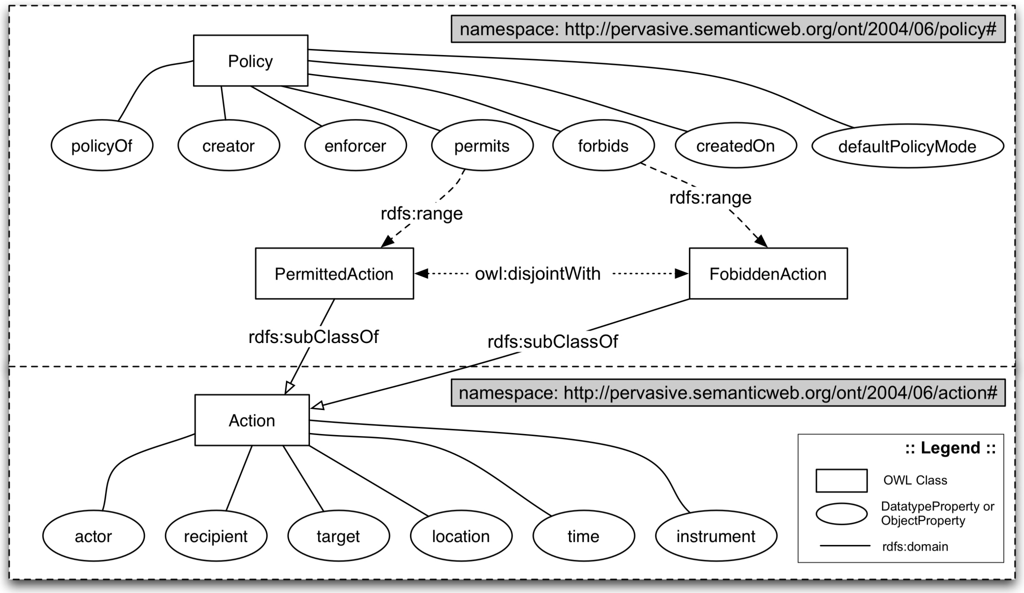


Figure 15: The SOUPA policy ontology defines the vocabularies for describing rules that permit or forbid agents to perform actions. The policy ontology imports the SOUPA action and time ontologies.

user. It would be infeasible for the user to explicitly enumerate location sharing rules for every agent in the system. A solution to this problem is to define meta-policy reasoning behavior that can help agents to reason about action permissions even when policy rules are not defined. The SOUPA policy ontology defines the following meta-policy reasoning behavior:

- **Conservative**: by default, assume all actions are forbidden. If no explicit rules are defined to regulate an action, then assume the action is forbidden.
- **Liberal**: by default, assume all actions are permitted. If no explicit rules are defined to regulate an action, then assume the action is permitted.

The OWL representation of the policy ontology is defined in a single ontology document under the XML namespace `http://pervasive.semanticweb.org/ont/2004/06/policy#`. This ontology imports the SOUPA time and action ontologies for representations of time and actions, respectively. Hereafter, I use the prefix `pol:`, `tme:`, and `act:` as the XML namespace shorthand for the SOUPA policy, time, and action ontology, respectively.

The OWL class `pol:Policy` represents a set of all policy documents. An individual of this class represents a policy document for regulating the permissions for agents to perform different actions. For a given `pol:Policy` class individual, the properties `pol:creator`, `pol:policyOf`, and `pol:enforcer` describe the agent who creates the policy, the agent whom this policy applies to, and the agent who enforces the defined policy rules, respectively. For describing the time when a policy document is created, the ontology defines the `pol:createdOn` property, and the range of which is the `tme:InstantThing` class. The individuals of the `tme:InstantThing` class are temporal descriptions of the time instants [24].

To describe policy rules, the ontology defines the property `pol:permits` for expressing permissions and the property `pol:forbids` for expressing prohibitions. The domains of both properties are the `pol:-Policy` class. The range of the `pol:permits` property is the `pol:PermittedAction` class, and the range of the `pol:forbids` property is the `pol:ForbiddenAction` class. Both action classes are subclasses of the `act:Action` class, but the set of individuals of each classes are disjointed.

The `pol:defaultPolicyMode` property is a meta-policy construct for specifying the policy reasoning behavior of a policy-enforcing agent. This property has domain `pol:Policy` and range `pol:Mode`. The `pol:Mode` class is an enumerated class, which consists of two pre-defined class individuals – `pol:-Conservative` and `pol:Liberal`.

The OWL class `act:Action` represents a set of all actions. Within the SOUPA action ontology, six basic action properties are defined. They are `act:actor`, `act:recipient`, `act:target`, `act:-location`, `act:time`, and `act:instrument`. The meanings of these properties are described in the previous section. Note that in the action ontology document, the ranges of these properties are unspecified, and they are intended to be specified in the subclasses that extend the `act:Action` class.

D. A Policy Reasoning Algorithm

This section describes an algorithm for reasoning with the SOUPA policy ontology. The algorithm describes a procedure that can be implemented by a self-governing agent to compute its right to perform a specific action or implemented by a centralized authority to answer queries about the rights of other agents to perform actions.

An agent capable of reasoning with the SOUPA policy ontology is assumed to have the following components: (i) $\mathcal{KB}$: a knowledge base that stores the agent's world knowledge in RDF triples (SOUPA ontologies, contextual information, user policies, etc.), (ii) $\mathcal{R}_{DL}$: a Description Logic inference engine that support the OWL-DL reasoning, (iii) $f_{action}(I)$: a function that maps an agent's intention I (i.e., an action that an agent intends to perform) into a set of RDF triples, and (iv) $f_{policy}(A)$: for a given action description A, this func-

Algorithm 2 Computes the permission to perform an intended action I

```
COMPUTE-PERMISSION($\mathcal{KB}$, $\mathcal{R}_{DL}$, $I$)
 1: $perm \leftarrow$ UNDECIDED
 2: $A \leftarrow f_{action}(I)$
 3: $P[] \leftarrow f_{policy}(A)$
 4: if $length[P] = 0$ then
 5:     $P[1] \leftarrow *$global_policy$*$
 6: end if
 7: for $i = 1$ to $length[P]$ do
 8:     $perm_{tmp} \leftarrow$ UNDECIDED
 9:     DO-CLASSIFICATION($\mathcal{R}_{DL}$, $\mathcal{KB}$, $P[i]$, $A$)
10:     $T[] \leftarrow$ LIST-INDIVIDUAL-CLASS-TYPES($\mathcal{R}_{DL}$, $A$)
11:     if CONTAINS($T$, PermittedAction) $=$ TRUE and
        CONTAINS($T$, ForbiddenAction) $=$ TRUE then
12:         error "inconsistent policy"
13:     else if CONTAINS($T$, PermittedAction) $=$ FALSE and
        CONTAINS($T$, ForbiddenAction) $=$ FALSE then
14:         $perm_{tmp} \leftarrow$ USE-POLICY-DEFAULT-BEHAVIOR($P[i]$)
15:     else if CONTAINS($T$, PermittedAction) $=$ TRUE then
16:         $perm_{tmp} \leftarrow$ PERMIT
17:     else if CONTAINS($T$, ForbiddenAction) $=$ TRUE then
18:         $perm_{tmp} \leftarrow$ FORBID
19:     end if
20:     if $perm_{tmp} =$ PERMIT and $perm =$ FORBID then
21:         error "inconsistent policy"
22:     else if $perm_{tmp} =$ FORBID and $perm =$ PERMIT then
23:         error "inconsistent policy"
24:     else
25:         $perm \leftarrow perm_{tmp}$
26:     end if
27:     FORGET-ASSERTIONS($\mathcal{R}_{DL}$,$P[i]$)
28: end for
29: return $perm$
```

```
USE-POLICY-DEFAULT-BEHAVIOR($policy$)
 1: $behavior \leftarrow$ POLICY-DEFAULT-BEHAVIOR($policy$)
 2: if $behavior =$ CONSERVATIVE then
 3:     return FORBID
 4: else if $behavior =$ LIBERAL then
 5:     return PERMIT
 6: end if
```

tion outputs a list of user policy documents that each of which contains some rule that regulates the execution of the action.

The outputs of the $f_{action}(I)$ and $f_{policy}(A)$ functions are expressed in the SOUPA ontologies or ontologies that extend SOUPA. Users are assumed to be responsible to provide agents their policy documents.

The objective of the algorithm is to help a policy-enforcing agent to effectively decide, for a given set of associated user policies and an intended action, whether the execution of the action is permitted or forbidden by the policies. Exception conditions may occur while the agent is performing policy reasoning. First, an exception can occur when there is no available user policy in the $\mathcal{KB}$. Second, an exception can occur when the rules in a user policy neither permit nor forbid the execution of an action. Lastly, an exception can occur when one or more user polices contain conflicting rules that simultaneously permit and forbid the execution of an action.

In this algorithm, the first exception is handled by assuming the existence of a global policy, which is adopted by the policy-enforcing agent by default. To handle the second exception, the algorithm uses the meta-policy construct (i.e., `pol:defaultPolicyMode`) to decide the permission of an action when it is not regulated by any policy rules. For the last exception, the algorithm delegates the exception handling task to the upper-level agent implementation by "flagging" the detection of policy inconsistency.

Algorithm 2 defines a procedure called COMPUTE-PERMISSION. It takes $\mathcal{KB}$, $\mathcal{R}_{DL}$ and I as the inputs and returns a value that represents the computed permission to perform an action A. Note that A is the output of $f_{action}(I)$. Let $\mathrm{P\_TYPE}$ be an enumeration type, which has a set of values PERMIT, FORBID, and $\mathrm{UNDECIDED}$. The values of $\mathrm{P\_TYPE}$ represent all possible computed permission values that can be returned by COMPUTE-PERMISSION.

In Algorithm 2, the procedure DO-CLASSIFICATION (line 9) is called to execute the associated DL classification functions of $\mathcal{R}_{DL}$, and the procedure LIST-INDIVIDUAL-CLASS-TYPES (line 10) is called to return the names of all ontology classes of which the action A is an instance after the classification is made. The utility procedure USE-POLICY-DEFAULT-BEHAVIOR (line 14) helps the agent to compute permissions when actions are not explicitly regulated by any policy rules. After computing the action permission for each user policy, the procedure FORGET-ASSERTIONS (line 27) is called to execute the associated DL functions to remove all assertions about the current policy from the inference engine, so that the inference of different policy documents can be computed independently.

E. Privacy Protection Applications

I prototyped a context broker to demonstrate privacy protection. In the prototype, the context broker implements the policy reasoning algorithm described in the previous section and is customized to support the protection of two types of private information: (i) personal profiles and (ii) location information. Personal profiles consist of information that describes a person's contact information, social networks, professional backgrounds, and personal devices. Location information is information that describes the whereabouts of a person.

The SOUPA and the CoBrA ontologies are used to express this information. The typical profile information is expressed using vocabularies from the SOUPA agent and person ontologies. The SOUPA space, time, and location ontologies are used to express people's location contexts. Since the prototype applications are built for a smart meeting scenario at UMBC, the CoBrA ontologies are used to express the information about the specific people, rooms, events, and services.

Using the CoBrA Demo Toolkit (see Chapter VII.), the prototype demonstrates the following privacy protection scenarios:

- A context broker has knowledge about a user's profile information (e.g., contact information or daily schedule). In the privacy policy, the user defines different classes of information recipients that are either permitted or forbidden to acquire this information. When an agent queries the context broker for the user's profile information, the context broker shares the matching information only if the policy permits the agent to receive this information.

- A context broker has knowledge about a user's location. This knowledge consists of information about the specific room that the user is located in and the associated arrival and departure time interval. When different agents queries the context broker for the user's location information, the context broker either reveals or hides the matching location information based on the privacy policy of the person.

- A context broker has knowledge about the different locations that a user has been to during a particular interval (i.e., the context broker has a history record of the user's location context). The specific locations described in the history record are a mixed set of spatial things (e.g, rooms, buildings, university campus, cities, states or countries). In the privacy policy, the user specifies the types of locations that are permitted or forbidden to be shared. When an agent queries the context broker for the user's location history, the context broker only shares the records that are permitted by the policy.

- A context broker has knowledge a user's location. The specific location described in this knowledge is a type of room. In the privacy policy, the user specifies the granularity of the location information that the context broker can share with other agents. For example, the policy rule might say "only share my location information if the described location is more general than the concept *building*". When an agent queries the context broker for the user's location information, the context broker will not reveal the name of the particular room in which the user is located. However, it will share the name of the building, the university campus, the state, or the country that spatially subsumes that particular room.

In the rest of this section, I describe an example of the privacy policy and the policy reasoning implementation of a context broker.

1. An Example of the Privacy Policy

A privacy policy is typically defined by importing the vocabularies from the SOUPA policy and action ontologies. Sometimes a privacy policy may also import specific classes (domain actions, geo-spatial concepts, etc.) from a domain ontology. The example policy described in this section imports the SOUPA and CoBrA ontologies vocabularies (see Chapter IV.).

The policy representation of the follow examples are expressed using the N3 language [11]. A full RDF/XML representation of the policy is available at `http://cobra.umbc.edu/ont/2004/05/harrychen-policy`.

Policy Rule 1

```
<http://umbc.edu/˜hchen4/hchen.pol> a pol:Policy;
  pol:policyOf  [ a per:Person; per:name "Harry Chen"ˆˆxsd:string ]
  pol:defaultPolicyMode pol:Conservative;

  # Rule 1: all individuals of CLS1 are permitted actions#
  pol:permits ha:CLS1;

ha:CLS1 a :Class;
   :intersectionOf (
      ebact:ShareContactInfo
      [ :allValuesFrom ha:PersonTimFininKnows; :onProperty act:recipient ] ) .

ha:PersonTimFininKnows a :Class;
   :intersectionOf (
      per:Person
      [ :hasValue <http://www.cs.umbc.edu/˜finin>; :onProperty per:knows ] ) .

      ...
```

The beginning of this policy defines the policy's owner and the default policy mode. Rule 1 expresses that action class `CLS1` is of type `pol:PermittedAction`. The context broker is permitted to perform actions that are instants of this class, e.g., sharing Harry's contact information with people that Tim Finin knows.

Action class `CLS1` is defined using `owl:intersection`, an OWL-DL construct for defining complex classes. The set of individuals of `CLS1` are in the class intersection between `ebact:ShareContactInfo` class and a restriction class. This restriction class has defined restriction on the property `act:recipient`, and which requires all range values of the property to be individuals of the class `PersonTimFininKnows`.

Policy Rule 2

```
<http://umbc.edu/˜hchen4/hchen.pol> a pol:Policy;
  pol:policyOf  [ a per:Person; per:name "Harry Chen"^^xsd:string ]
  pol:defaultPolicyMode pol:Conservative;

  ...

  # Rule 2: all individuals of CLS2 are permitted actions#
  pol:permits ha:CLS2;

ha:CLS2 a :Class;
   :intersectionOf  (
      ebact:ShareLocationInfo
      [ :allValuesFrom ebm:EbiquityMember; :onProperty act:recipient ]
      [ :onProperty act:target; :someValuesFrom ha:MyRestrictedLocationContext ] ) .

ha:MyRestrictedLocationContext a :Class;
  :intersectionOf  (
     loc:LocationContext
     [ :onProperty loc:boundedWithin; :someValuesFrom ha:foo-a1 ] ) .

ha:foo-a1 a :Class;
  :oneOf  (ebgeo:ITE210A  ebgeo:ITE325B ebgeo:UMBCMainCampus) .

  ...
```

Rule 2 expresses a different class of actions that the context broker is permitted to perform. This rule gives the context broker the permission to share Harry's location information with the members of the eBiquity research group. However, the policy restricts the kind of location information can be shared with the eBiquity group members. Information can be shared if it only reveals that Harry is located in any one of the following places: Room ITE201A, Room ITE325B, or UMBC Main Campus.

Like the `CLS1` class, the `CLS2` class is also defined as an intersection class. `CLS2` is a set of individuals that are in the class intersection between `ebact:ShareLocationInfo` and two other restriction classes. One of the restriction classes has defined restriction on the property `act:recipient`, and which requires all of the range values point to the `ebm:EbiquityMember` class. The other restriction class has defined restriction on the property `act:target`, and which requires some of the range values point to the `foo-a1` class. The `foo-a1` class is an enumerated class, which consists of three different `spc:SpatialThing` individuals defined in the CoBrA `ebgeo` namespace.

Policy Rule 3

```
<http://umbc.edu/˜hchen4/hchen.pol> a pol:Policy;
```

```
  pol:policyOf  [ a per:Person; per:name "Harry Chen"^^xsd:string ]
  pol:defaultPolicyMode pol:Conservative;

  ...

  # Rule 3: all individuals of CLS3 are forbidden actions#
  pol:forbids ha:CLS3.

ha:CLS3 a :Class;
   rdfs:comment "Share my location information with untrusted service agent";
   :intersectionOf  (
      ebact:ShareLocationInfo
      [ :allValuesFrom ha:UntrustedServiceAgent; :onProperty act:recipient ] ) .

ha:UntrustedServiceAgent a :Class; rdfs:subClassOf agt:Agent;
    :oneOf  (
        <http://www.orbitz.com#locTrack> <http://www.foobar.com#whereRu>
        <http://www.foofoobar.com#abc>  ) .

   ...
```

Rule 3 expresses a policy rule that forbids the context broker to share Harry's location information with a group of untrusted agents. The `CLS3` class is a set of actions that the context broker is forbidden to perform.

`CLS3` is defined in a similar way as `CLS1` and `CLS2`. This class is an intersection class between `ebact:ShareLocationInfo` and a restriction class. The restriction class has defined restriction on the property `act:recipient`, and which requires all of the range values point to the `UntrustedService-Agent` class. `UntrustedServiceAgent` is an enumeration class of three different `agt:Agent` individuals.

The interpretation of this rule is that if any of the three agents asks the context broker for Harry's location, regardless of the granularity of the location information, the context broker will always refuse to share this information.

Policy Rule 4

```
<http://umbc.edu/~hchen4/hchen.pol> a pol:Policy;
  pol:policyOf  [ a per:Person; per:name "Harry Chen"^^xsd:string ]
  pol:defaultPolicyMode pol:Conservative;

  ...

  # Rule 4: all individuals of CLS4 are permitted actions#
  pol:permits ha:CLS4

ha:CLS4 a :Class;
   :intersectionOf (
      ebact:ShareLocationInfo
      [ :allValuesFrom per:Person; :onProperty act:recipient ]
      [ :onProperty act:target; :someValuesFrom ha:MyRestrictedLocationContext2 ] ) .

ha:MyRestrictedLocationContext2 a :Class;
  :intersectionOf  (
     loc:LocationContext
     [ :onProperty loc:boundedWithin; :someValuesFrom ha:PlaceMoreGeneralThanITE ] ) .

ha:PlaceMoreGeneralThanITE a :Class;
   :intersectionOf (
      rcc:RCCSpatialRegion
      spc:LandBasedRegion
      [ onProperty spc:spatiallySubsumes; :allValuesFrom ebgeo:ITE ] ) .
```

Rule 4 expresses a policy rule that permits the context broker to share Harry's location with any person, however, a special restriction is imposed on the location granularity. The policy rule permits Harry's location information to be shared if the information only reveals that Harry is located in a place that is no more specific than the ITE building – e.g., it is okay to reveal that Harry is on the UMBC campus, but it is not okay to reveal that Harry is located in a particular room in the ITE building.

The `CLS4` class is also defined as an intersection class. This class is in the intersection between `ebact:-ShareLoocationInfo` and two other restriction classes. One of the restriction classes has defined restriction on the property `act:recipient`. The restriction requires all of the range values of the `act:-recipient` property point to the `ebm:EbiquityMember` class. The other restriction class has defined restriction on the property `act:target`. The restriction requires some of the range values point to the `MyRestrictedLocationContext2` class.

The `MyRestrictedLocationContext2` class is a class that represents a special type of location context. The individuals of this class are `loc:LocationContext` individuals with a restricted `loc:-boundedWithin` property. This property restriction requires some of the range values point to the `Place-MoreGeneralThanITE` class.

2. Policy Reasoning Walk-Through

The policy reasoning in the context broker prototype has two parts. One part is an implementation of the policy reasoning algorithm (i.e., Algorithm 2) for computing the permissions to perform actions. The other part is a procedure for adjusting information granularity. The implementation requires all policies for privacy protection and all contextual information involved in knowledge sharing to be expressed in the OWL language.

Policies and information are communicated between the context broker and the other agents via FIPA ACL. The content message representation is expressed in RDF/XML. To acquire contextual information, agents send query messages to the context broker. The contents of those query messages are expressed using the RDQL language.

The context broker stores the user-defined policies in the same persistent knowledge base that stores the users' contextual information (see Chapter V.). Since all policies are represented as RDF statements, the implementation manages these policies the same way as it manages the contextual knowledge. The Jena Ontology API is used to manipulate and access the stored user policies.

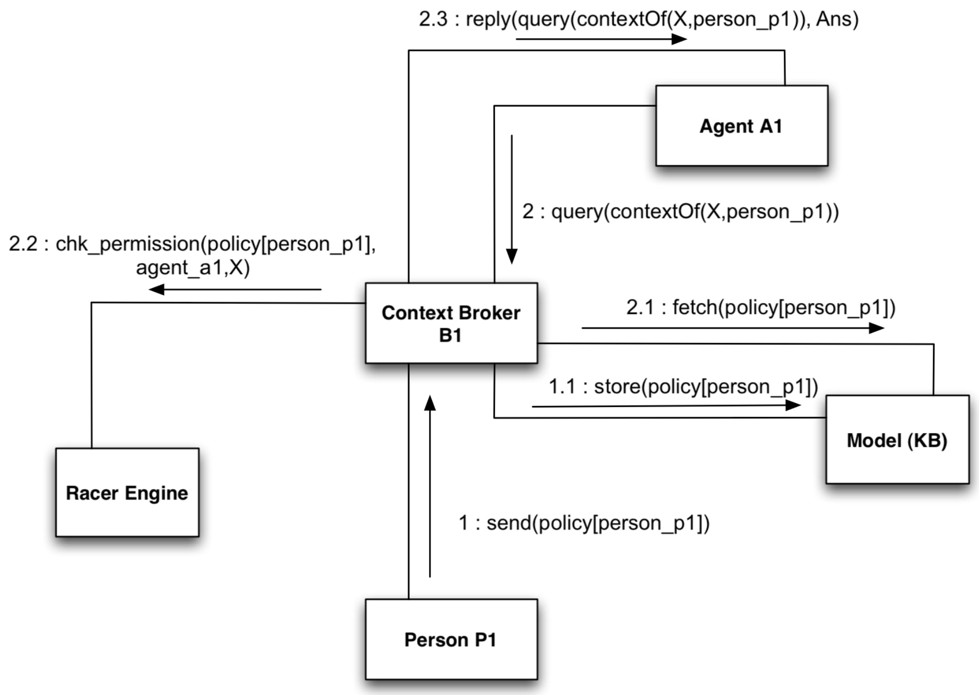

2.3 : reply(query(contextOf(X,person_p1)), Ans)
Agent A1
2 : query(contextOf(X,person_p1))
2.2 : chk_permission(policy[person_p1], agent_a1,X)
Context Broker B1
2.1 : fetch(policy[person_p1])
1.1 : store(policy[person_p1])
Model (KB)
Racer Engine
1 : send(policy[person_p1])
Person P1

Figure 16: The interactions between the context broker and other agents during a privacy protection process. Before the context broker shares any person's information with the other agents, it uses the person's policy to compute whether or not the information can be shared.

To reason about the policies, the implementation uses the Racer inference engine [48]. The context broker does not directly invoke the DL inference commands of Racer. It uses the DIG inference API provided by Jena to interact with Racer. In Jena, all ontology classes and instances are stored in a class object called `Model`. An instance of this class can be viewed as a database of the context broker. By itself, the `Model` class does not provide any ontology inference support. To reason with the stored information, a reasoner must be attached to the `Model` object. When a reasoner is used, all queries sent to the `Model` object are automatically processed by the underlying inference procedures of the reasoner. In the Jena API, there is a special reasoner called `DIGReasoner`, which provides DL inference support by loading data into an external Racer engine and calling the appropriate DL inference commands over Racer. The context broker implementation uses this reasoner to support policy reasoning.

The Interactions in Privacy Protection

In the prototype demonstration, the privacy protection of a context broker involves interactions between multiple agents. Figure 16 shows a UML collaboration diagram of these interactions. Key actors in the demonstration are context broker B1, information seeking agent A1, and person P1. The demonstration assumes context broker B1 has some contextual knowledge about person P1, and agent A1 seeks to acquire this information. To protect her privacy, person P1 sends a policy to context broker B1 and requests the context broker to enforce the defined policy rules when sharing her contextual information. When context broker B1 receives a query from agent A1, requesting to acquire some contextual information about person P1, context broker B1 consults person P1's privacy policy before revealing any information to agent A1.

When building the prototype system, there are two implementation issues: (i) How to match a correct user policy when the information resulted from agent A1's query involves multiple people? (ii) What is the appropriate message to send back to agent A1 when person P1's policy forbids the matched query result to be shared?

The solution adopted in the prototype implementation is as the follows. When information resulted from a query involves multiple people, the policies of all individuals are used to compute whether or not the information can be shared. For example, if the result of a query describes that Harry Chen is meeting with Tim Finin in Room 232. Both Harry Chen's and Tim Finin's policies are used to decide whether or not this information can be reveal to the requesting agent.

According the FIPA Query Interaction Protocol [43], when an agent decided not to share some information with an information requesting agent, the agent should reply with a FIPA "refuse" message in response to the request. In other words, according to the protocol, when context broker B1 refused to share person P1's information with agent A1, it should reply with a "refuse" message. However, in the context of privacy protection, the sending of this response message is inappropriate because it may cause undesired information leakage. For example, let's assume context broker B1 knows that Harry Chen is located in RM 233, and agent A1 asks context broker B1 "what's the location of Harry Chen?" Let's assume Harry's policy forbids his location information to be shared with agent A1, and for that reason, context broker B1 replies with a "refuse" message. Although no explicit location information is revealed to agent A1, some implicit location information can be deduced through reasoning. If agent A1 knows context broker B1 is located on the UMBC campus and it only maintains people's location information within the vicinity of the campus, agent A1 can conclude Harry Chen is located somewhere on the UMBC campus. To address this issue, the "refuse" communication act is replace by a new communication act called "uncertain". An informal semantic description of this communication act is the following: the "uncertain" act is performed when the agent does not have sufficient knowledge to answer a received query. Revealing a context broker does not have sufficient knowledge to answer a query will prevent other agents from exploiting the hidden knowledge that may be associated with the “refuse” communication action.

Adjusting the Granularity of Information

The prototyped context broker can also support the adjustment of information granularity when protecting privacy. User policies may define rules to control the granularity of the information that the context broker is permitted to share. For example, Rule 4 described in the previous section is an example of this type of policy rule. In this rule, the policy permits the context broker to share Harry’s location information with any person as long as the information only reveals that Harry is located in a place that is more general than the concept of “building”.

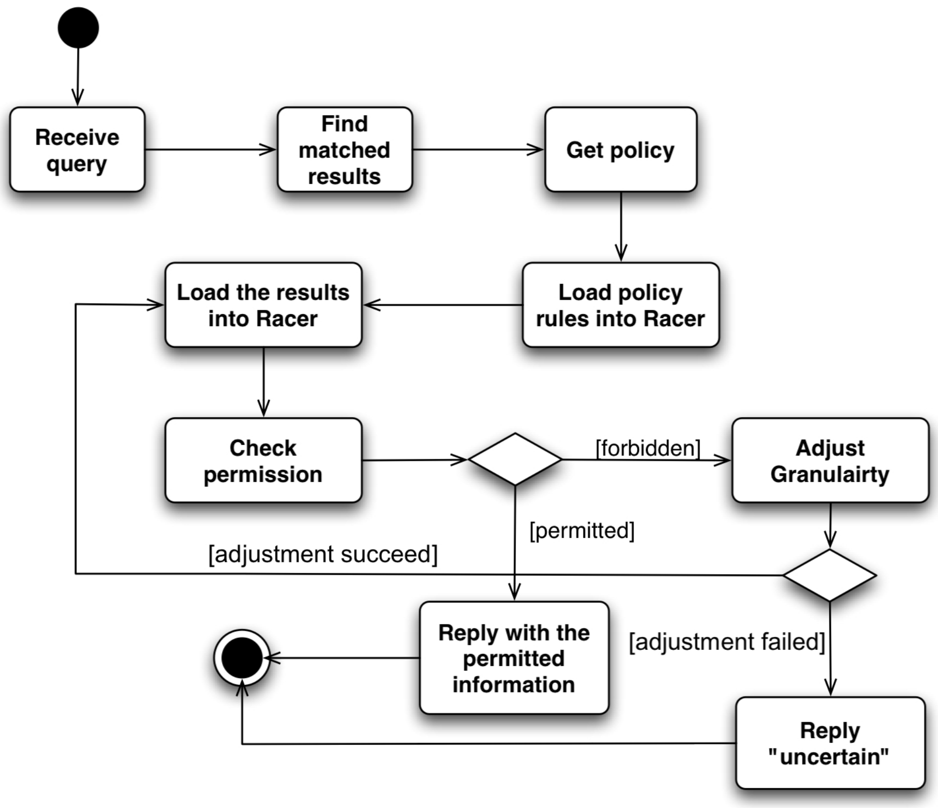


Figure 17: A UML diagram that describes the context broker’s behavior when adjusting information granularity for privacy protection.

The granularity adjustment procedure is implemented based on an algorithm described in Figure 17. This algorithm extends the policy reasoning algorithm described previously with additional agent behaviors. The algorithm begins when the context broker receives a query from another agent. After executing the query (which is represented in RDQL), the context broker finds the matched results (which is represented in

RDF/XML). From the query results, the context broker determines the policies that should be used in privacy protection. After loading the policies and the query results into the Racer engine, the context broker computes whether or not the sharing of this information is permitted. If it is permitted, then the context broker shares this information. Otherwise, the context broker tries to adjust the granularity of the information. If the granularity of the information can be successfully adjusted, then the context broker re-computes the permission. Otherwise, the context broker sends an "uncertain" message to the sender agent and the granularity adjustment procedure terminates.

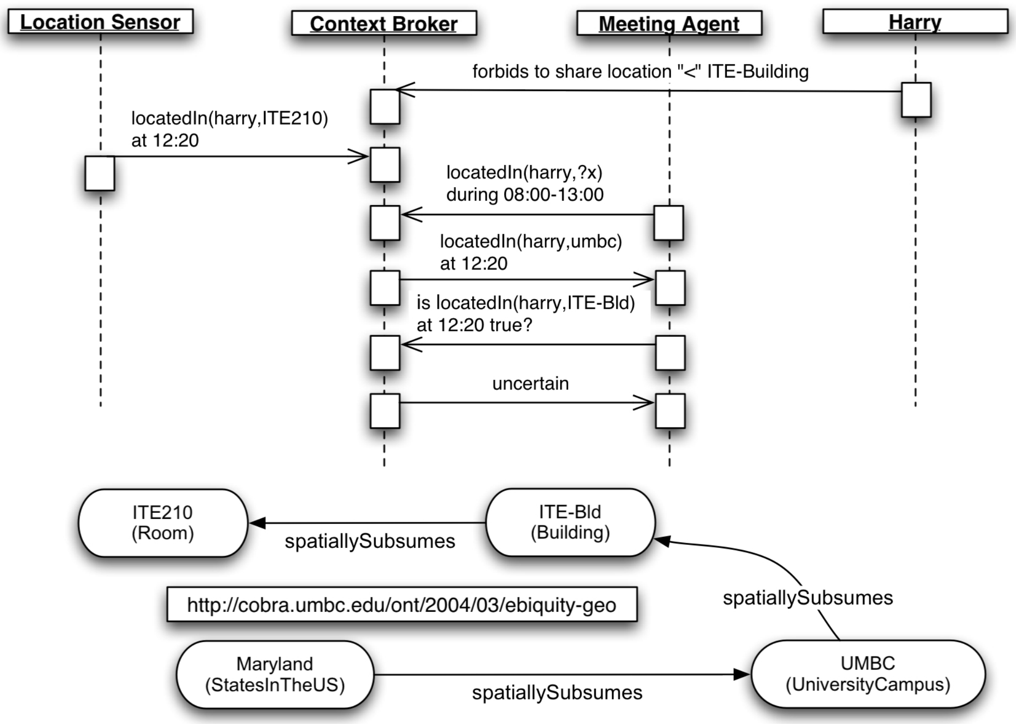

Location Sensor
Context Broker
Meeting Agent
Harry
forbids to share location "<" ITE-Building
locatedIn(harry,ITE210) at 12:20
locatedIn(harry,?x) during 08:00-13:00
locatedIn(harry,umbc) at 12:20
is locatedIn(harry,ITE-Bld) at 12:20 true?
uncertain
ITE210 (Room)
spatiallySubsumes
ITE-Bld (Building)
spatiallySubsumes
http://cobra.umbc.edu/ont/2004/03/ebiquity-geo
Maryland (StatesInTheUS)
spatiallySubsumes
UMBC (UniversityCampus)

Figure 18: The context broker exploits spatial reasoning to generalize the actual location of a person and shares this information without violating the user defined policy.

Figure 18 shows an example of the granularity adjustment procedure. In this example, the context broker exploits the use of a spatial ontology to adjust the granularity of Harry's location information. Harry's policy forbids any location information that is more detail than the concept of "ITE Building" to be shared. When the Location Sensor tells the context broker that Harry is located in Room ITE210, based on the defined spatial ontology, the context broker is able to conclude additional location information about Harry's location (e.g., Harry is also located in ITE Building, UMBC Campus, and the state of Maryland). When the Meeting Agent asks for the whereabouts of Harry during 8:00-13:00, the context broker replies that Harry is located in UMBC. It does not reveal the more specific knowledge that it has about Harry’s location – i.e., Harry is located in ITE210. When the Meeting Agent asks whether or not Harry is located in the ITE Building at 12:20, the context broker replies with an “uncertain” message.

Chapter VII.

IMPLEMENTATIONS

To evaluate the feasibility of the Context Broker Architecture, I have developed a number of different prototype systems to simulate the use of a context broker in a smart meeting room environment. The objective is to show (i) the use of CoBrA can help to reduce the efforts in developing a pervasive context-aware system, (ii) a context broker can be implemented based on the existing FIPA agent and semantic web technologies, and the tools for testing and monitoring a context broker can be integrated into the mainstream software development environment (e.g., Eclipse IDE), and (iii) new context-aware applications can be built on the existing resource-limited mobile devices by exploiting the broker-centric design of CoBrA.

A. EasyMeeting

EasyMeeting is an extension to Vigil [106], a third generation pervasive computing infrastructure developed at UMBC. Security is the main focuses in Vigil. While the research development behind Vigil shows great promisespromise in building flexible and secure smart spaces [106, 26], it lacks the necessary support for context-awareness and privacy protection [22]. To improve upon the previous system, a context broker is added to provide support for context modeling and reasoning (see Figure 19). The development of EasyMeeting was a join work between me and a number of other eBiquity group members [23].

1. Services

EasyMeeting's implementation provides context-aware services for assisting speakers and audiences during presentations. We have developed sixedsix services:

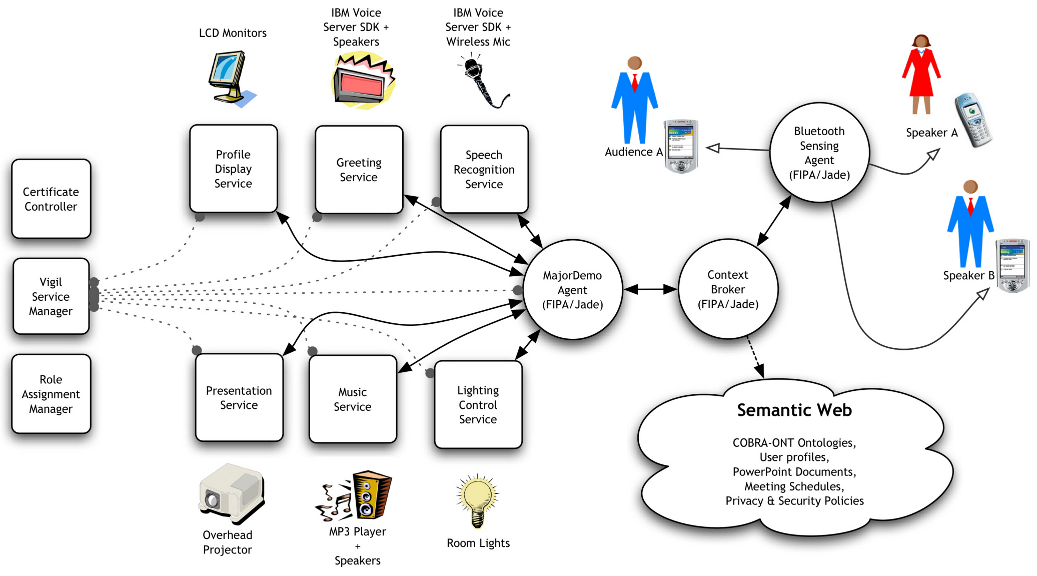

LCD Monitors
IBM Voice Server SDK + Speakers
IBM Voice Server SDK + Wireless Mic
Certificate Controller
Profile Display Service
Greeting Service
Speech Recognition Service
Vigil Service Manager
Role Assignment Manager
MajorDemo Agent (FIPA/Jade)
Context Broker (FIPA/Jade)
Bluetooth Sensing Agent (FIPA/Jade)
Audience A
Speaker A
Speaker B
Presentation Service
Music Service
Lighting Control Service
Semantic Web
COBRA-ONT Ontologies, User profiles, PowerPoint Documents, Meeting Schedules, Privacy & Security Policies
Overhead Projector
MP3 Player + Speakers
Room Lights

Figure 19: In EasyMeeting the Context Broker shares its contextual knowledge with the MajorDemo agent. Using this knowledge, MajorDemo selects and then invokes appropriate Vigil services to provide relevant services and information to the speakers and audiences.

1. **Speech Recognition Service** This service can be invoked to perform speech recognition on a set of pre-defined voice input vocabularies (e.g., “yes”, “no”, “show Harry’s presentation”) and generate CCML (Centaurus Capability Markup Language) [59] commands for controlling other Vigil services. In this service, the underlying voice recognition procedure is implemented using the IBM WebSphere Voice Server SDK and Voice XML.

2. **Presentation Service** This service can be invoked to display PowerPoint presentations on an overhead projectprojector in the room. The presentation file is fetched from a URL that a client has specified. It defines a set of CCML commands for controlling the flow of a displaying presentation (e.g., “next”, “back”, “stop”, “start”). Exploiting the Speech Recognition Service, users can control their presentations via speech commands.

3. **Lighting Control Service** This service can be invoked to adjust the lighting conditions in a meeting room. The underlying lighting control mechanism is implemented using the X10 technology. Lights are wired to X10 Lamp Modules. Using the light control API, this service can control the lights to be turned on or off, or to be dimmer or brighter.

4. **Music Service** Connected to a pair of speakers and using an existing MP3 music player software, this service can be invoked to play audio music files that are accessible on the Web. A typical use of this

service is to continuously play background music while participants are waiting for a meeting to begin.

5. **Greeting Service** This service can be invoked to play a specified greeting message. The played audio file is dynamically generated from a client specified greeting message (e.g., "Welcome to the eBiquity Group, President Hrabowski"). The underlying text-to-speech procedure is implemented using the IBM WebSphere Voice Server SDK and Voice XML.

6. **Profile Display Service** This service can be invoked to instruct all subscribed web browsers to display any URL (e.g., some user's home page). The intended use of this service for displaying speaker profiles on the handheld devices that individual audiences carry. This is useful for helping audiences to learn about the background of the speaker that is currently presenting.

In EasyMeeting, the context broker provides a shared model of context for all agents and services. In particular, it is responsible for acquiring and maintaining consistent knowledge about (i) the location of meeting participants, (ii) the event schedule of a meeting, (iii) the presentations that are scheduled for the meeting, (iv) the profiles of the presentation speakers, and (v) the state of a meeting. To acquire this knowledge, the context broker explores different sources of information that is published on the Semantic Web and that is provided by the sensor agents (e.g., the Bluetooth Sensing Agent).

The role of a MajorDemo agent is to decide when and what services should be provided to the meeting participants. It relies on the context broker to provide information about the meeting context and uses the registered Vigil services to facilitate different meeting related tasks. In order to simultaneously interact with the Vigil services and the context broker, this agent is implemented with a hybrid design that bridges the API's for invoking services in Vigil and for communicating with the agents in CoBrA.

2. Use Case

The following is a typical EasyMeeting use case: Room 338 is a smart meeting room. On January 8th, 2004, a presentation is scheduled to take place from 1:00-2:30 PM in this room. Moments before the event starts, the room's Context Broker acquires the meeting's schedule from the Semantic Web and concludes the meeting is about to take place in the Room 338. As the meeting participants begin to arrive, the room's Bluetooth Sensing Agent detects the presences of different Bluetooth enabled devices (e.g., cellphones, PDA's). Because each device has a unique device profile that is represented in some standard device ontologies, the sensing agent can share this information with the Context Broker.

Based on the user profile information stored in the knowledge base of the context broker (e.g., who owns what devices), without knowing any evidence to the contrary, the context broker concludes the owners of the detected devices are also located in the Room 338. Among the arrived participants, Harry the speaker and President Hrabowski the distinguished audience are two people that are listed in the meeting schedule. The Context Broker shares the location information of these listed participants with the subscribed MajorDemo agent.

Knowing that President Hrabowski has a distinguished audience role, the MajorDemo agent invokes the Greeting Service to greet him. At 1:00 PM, the Context Broker informs the MajorDemo agent that all listed *key* participants have arrived and that the presentation can start. Knowing all the lights in the meeting are currently switched on and the background music is also playing, the agent invokes the Dim Light Method on the the Light Control Service and the Stop Music Method on the Music Service.

As Harry walks to the front of the meeting room, he speaks to the system using a wireless microphone, “load Harry’s presentation”. The voice command is received by the Voice Recognition Service and a corresponding CCML command is generated. The MajorDemo agent sends this text string command to the Presentation Service along with the URL at which Harry’s presentation can be downloaded (this information is provided by the Context Broker). As the Presentation Service loads Harry’s PowerPoint slides, the MajorDemo agent invokes the Profile Display Service to show Harry’s home page. A few seconds later, all LCD displays sitting on the conference table start showing Harry’s biosketch and his profile. Using the same wireless microphone, Harry speaks to the system to control his presentation.

B. CoBrA Demo Toolkit

CoBrA Demo Toolkit is a set of software applications for demonstrating the key features of a context broker. Building on the existing FIPA and semantic web technologies, this toolkit showcases the use of ontologies for context modeling and knowledge sharing, the use of logical inference for context reasoning, and the use of policy for privacy protection.

This toolkit has three key components: (i) Context Broker – a context broker prototype built on the JADE agent library, (ii) ScriptPlay – a configurable JADE agent that facilities the CoBrA demo scripts, and (iii) CoBrA Eclipse Viewer (CEV) – an Eclipse Plug-in for monitoring the Context Broker’s knowledge base and behavior.

1. Context Broker Agent

The implementation of the Context Broker agent consists of 63 Java class files. The core behavior of this agent is defined in a set of Java classes called "behavior classes". These classes are subclasses of the abstract JADE agent behavior classes (`SimpleBehaviour`, `CyclicBehaviour`, etc.) [6]. The behavior classes define how the agent publishes its services, acquires contextual information from sensors and the Web, reasons about context and policy, and shares knowledge with other agents.

All behavior classes are contained within a single JADE agent class (i.e., ContextBroker), which is a subclass of the `jade.core.Agent` class. In this class, the behaviors are registered with an internal behavior scheduler. This scheduler manages the run-time execution of different behavior. Through the `ContextBroker` class, the behavior classes share a common set of resources (i.e., knowledge base, reasoners, and other miscellaneous data structures).

The knowledge base of the Context Broker agent is a Jena ontology model [18]. It is backed by a MySQL database. The knowledge stored in this model is represented as RDF triples. The Jena API provides the necessary API for maintaining this ontology model, as well as the API for querying the ontology model. Different ontology reasoners can be attached to the knowledge base (e.g., RDF-S and OWL reasoners). Most reasoning behaviors of the Context Broker agent use the Jena OWL reasoner for reasoning over the OWL language semantics. Some others use customized Jena rule-based reasoners to reason over ontology semantics that are not part of the OWL language (e.g., the SOUPA time and spatial ontologies).

Some behavior classes also usesuse external rule-based systems (e.g., Jess and Prolog) for context reasoning. In these reasoning implementations, the behaviors fetch the RDF triples from the knowledge base and transform them into the native representations of the rule-base systems. Some transformation procedures are implemented based on a set of ad-hoc rules, and some others exploit the use of XSLT (XSL Transformations) technology. For example, the XSLT technology is used in transforming the RDF representation of the contextual information into Theorist formulaesformulas when a reasoning behavior needs to use the Theorist system to resolve inconsistent information.

The privacy protection behavior uses the Racer reasoner for reasoning with policy rules. The behavior exploits the Jena API to communicate with the Racer reasoner via the DIG inference [55]. When reasoning about a person's privacy policy, the behavior fetches the policy rules from the knowledge base and loads the rules into a specially constructed Jena model that is attached with a DIG reasoner. Action instants and other required ontologies are also loaded into the model. Through DL classification, the behavior decides whether a given action instant is permitted or forbidden to perform.

The Context Broker implementation uses the FIPA ACL as the standard language for communication. The toolkit has a pre-defined set of communication protocols for knowledge sharing and action executions. The content of all knowledge sharing messages are expected to be expressed in the RDF/XML language, and the content of all query messages are expected to be expressed in RDQL language. The SOUPA and CoBrA ontologies are the standard ontologies for describing information in the message content.

The implementation provides a Java property file for configuring various run-time parameters of the agent. These include the service description used to publish services in a local FIPA Directory Facilitator, ontologies used in context reasoning, and administrative information for access the local MySQL database.

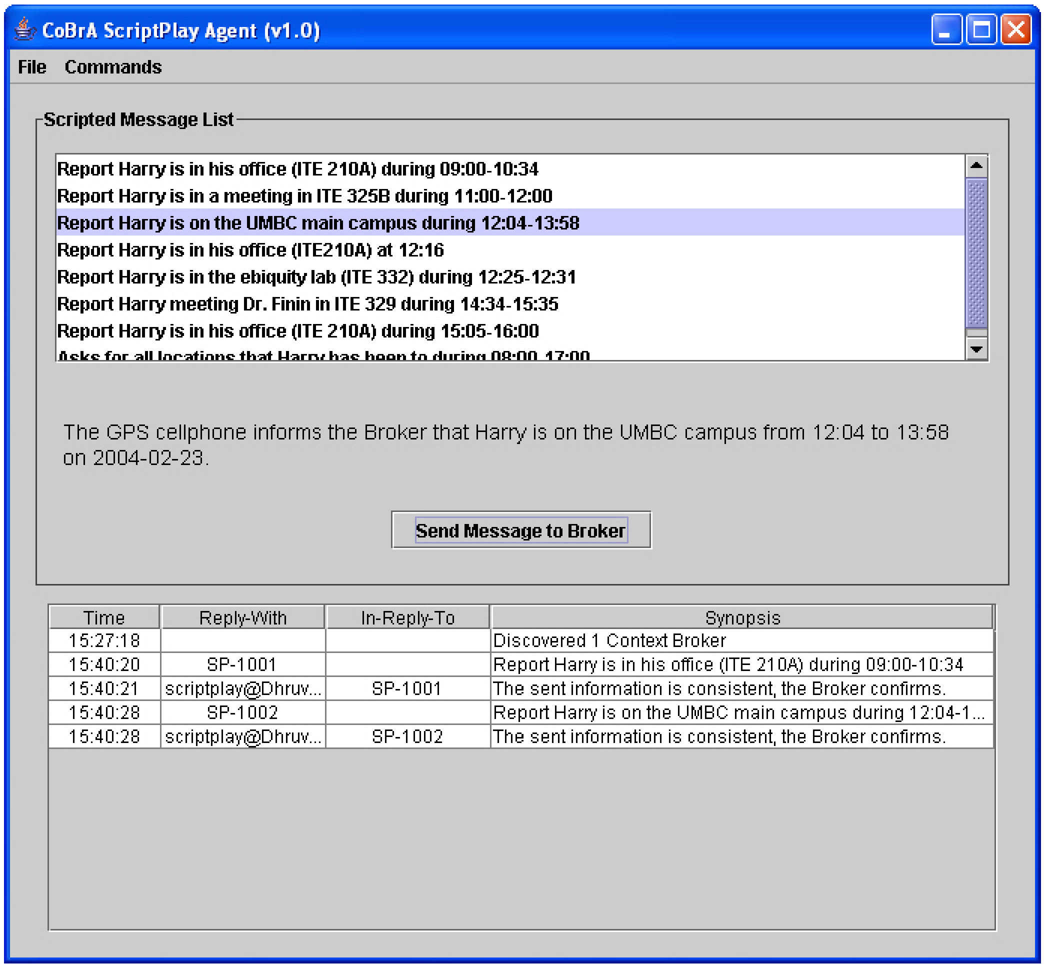
2. ScriptPlay Agent

The implementation of the ScriptPlay agent consists of 21 Java class files. Like the Context Broker agent's implementation, the ScriptPlay agent's behavior is also defined in a set of JADE agent behavior classes. These behaviors are registered with the internal scheduler of the `ScriptPlayAgent` class, which extends the `jade.core.Agent` class.

A key function of this agent is to allow developers to introspectinspect? the underlying behavior of a context broker. This includes examining the correctness in logical inference and privacy protection, inspecting the accurate representation of various contextual knowledge, and the proper enforcement in privacy protection. Through different introspectionmight not be the word you want, check the dictionary processes, the ScriptPlay agent helps the developers to understand different runtime behaviors of a context broker.

An introspection process is carried out in a sequence of message exchange between the Context Broker agent and the ScriptPlay agent. There are three kinds of communication messages: inform messages, query messages, and control messages. The inform messages are messages that contain information that the ScriptPlay wants to share with the Context Broker agent (e.g., information about a person's location or a meeting schedule). The query message are messages that contain queries that the ScriptPlay wants the Context Broker agent to answer (e.g., what'swhat is the location context of Harry during 8-9 AM?). The control messages are messages that the ScriptPlay agent sends to request the Context Broker to perform certain actions (e.g., clear your current knowledge base).

The communication messages used in different introspection processes can be grouped into an XML script file. Each script file can be viewed as a configuration file for the ScriptPlay agent. Depending on



Report Harry is in his office (ITE 210A) during 09:00-10:34
Report Harry is in a meeting in ITE 325B during 11:00-12:00
Report Harry is on the UMBC main campus during 12:04-13:58
Report Harry is in his office (ITE210A) at 12:16
Report Harry is in the ebiquity lab (ITE 332) during 12:25-12:31
Report Harry meeting Dr. Finin in ITE 329 during 14:34-15:35
Report Harry is in his office (ITE 210A) during 15:05-16:00
Asks for all locations that Harry has been to during 08:00-17:00

The GPS cellphone informs the Broker that Harry is on the UMBC campus from 12:04 to 13:58 on 2004-02-23.

Time	Reply-With	In-Reply-To	Synopsis
15:27:18			Discovered 1 Context Broker
15:40:20	SP-1001		Report Harry is in his office (ITE 210A) during 09:00-10:34
15:40:21	scriptplay@Dhruv...	SP-1001	The sent information is consistent, the Broker confirms.
15:40:28	SP-1002		Report Harry is on the UMBC main campus during 12:04-1...
15:40:28	scriptplay@Dhruv...	SP-1002	The sent information is consistent, the Broker confirms.

Figure 20: A screenshot of the ScriptPlay agent's GUI interface. Through this GUI, a user can select to use different introspection script files, step through an introspection process, and review the communication messages exchanged between the ScriptPlay agent and the Context Broker.

the script file that is loaded into the agent, the ScriptPlay agent dynamically configures itself to support the corresponding introspection process.

The current implementation provides a number of different introspection script files. They include the following:

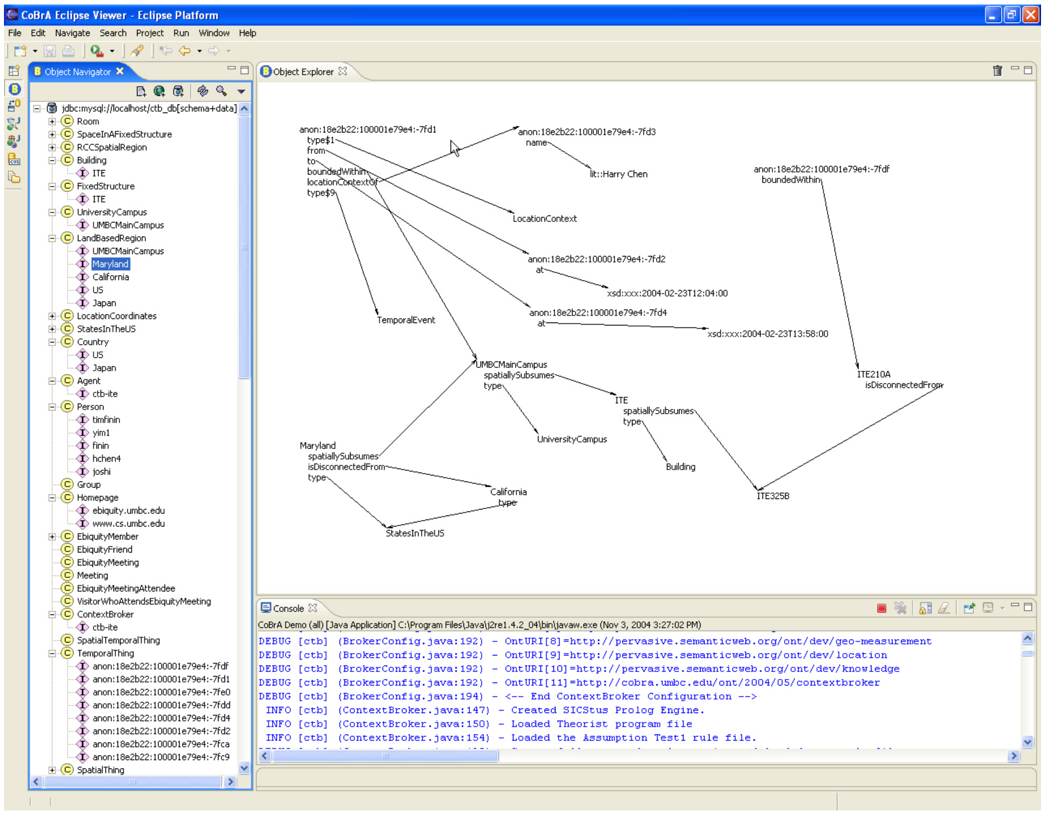
- A script file for examining the temporal reasoning of the Context Broker. It tests if the agent can correctly infer various types of temporal relations associated with time intervals and time instants.
- A script file for examining the spatial reasoning of the Context Broker. It tests if the agent can correctly infer the spatial subsumption relations between different spatial entities. The spatial entities may be expressed in either geo-metric or symbolic representations.
- A script file for examining the policy reasoning of the Context Broker. It tests if the agent can correctly enforce the user-defined privacy policy when sharing a person's location context or profile information.
- A script file for testing the Context Broker's ability to detect inconsistent location information about a person or a device. A set of location context information is sent to the agent with conflicting spatial and temporal descriptions. It tests if the agent can correctly detect inconsistency based on spatial and temporal reasoning.

For user interactions, the ScriptPlay agent provides a graphical user interface (GUI). Through this GUI, a user can change the introspection script file to be used, step through the introspection process, and review communication messages exchanged between the ScriptPlay agent and the Context Broker. Figure 20 shows a screen shot of this GUI interface.

3. CoBrA Eclipse Viewer

CoBrA Eclipse Viewer is an Eclipse Plug-in for monitoring the Context Broker's knowledge base and behavior. CEV also can be used as an ontology browser for exploring ontologies expressed using the OWL language. Ontologies may be stored on a local file system, a remote web server, or a persistent relational database system. The CEV implementation was inspired "the Spider for Eclipse" program developed by Kent Beck and Erich Gamma (`http://www.javaspider.org/`).

The GUI of CEV consists of three view panels (see Figure 21): Object Navigator, Object Explorer, and Console. In the Object Navigator view, the user can browse an opened knowledge base in a tree-view



```
CoBrA Demo (all) [Java Application] C:\Program Files\Java\j2re1.4.2_04\bin\javaw.exe (Nov 3, 2004 3:27:02 PM)
DEBUG [ctb] (BrokerConfig.java:192) - OntURI[8]=http://pervasive.semanticweb.org/ont/dev/geo-measurement
DEBUG [ctb] (BrokerConfig.java:192) - OntURI[9]=http://pervasive.semanticweb.org/ont/dev/location
DEBUG [ctb] (BrokerConfig.java:192) - OntURI[10]=http://pervasive.semanticweb.org/ont/dev/knowledge
DEBUG [ctb] (BrokerConfig.java:192) - OntURI[11]=http://cobra.umbc.edu/ont/2004/05/contextbroker
DEBUG [ctb] (BrokerConfig.java:194) - <-- End ContextBroker Configuration -->
 INFO [ctb] (ContextBroker.java:147) - Created SICStus Prolog Engine.
 INFO [ctb] (ContextBroker.java:150) - Loaded Theorist program file
 INFO [ctb] (ContextBroker.java:154) - Loaded the Assumption Test1 rule file.
```

Figure 21: A screenshot of the CoBrA Eclipse Viewer. CEV can be used to monitor the Context Broker's knowledge base and behavior. It also can be used as an ontology browser for exploring ontologies expressed using the OWL language.

structure. The nodes in the tree represent different ontology concepts (i.e., OWL classes and class instants). By double clicking on a class instant node (i.e., any node with an "I" icon), the user can explore its associated ontology properties in the Object Explorer view.

In the Object Explorer view, the ontology concepts are displayed as a graph with directed arcs. The graph structure is similar to the RDF graph [70]. The nodes in the graph represent RDF resources, and each node can be viewed as a subject or an object resource. The associated predicates of a subject resource are listed right below the subject node. The predicates are linked to the associated object resources via direct arcs. For example, in Figure 21, the node "Maryland" is a subject resource and the text labels listed right below the node are the associated predicate resources. The predicate "isDisconnectedFrom" is linked to the object node "California" via a directed arc.

The Console view is a logging window for monitoring the Context Broker's behavior. For debugging and inspecting the internal reasoning and execution of the agent, this view shows the logging messages that are produced by the logging thread of the Context Broker. The underlying logging function is supported by the Log4J logging system (`http://logging.apache.org`).

C. CoBrA Text Messaging Commands

CoBrA Text Messaging Commands (CTMC) is a text messaging interface for mobile users to interact with a Context Broker. Built on the existing text messaging technology, the users can use cellphones to ask the Context Broker questions and invoke actions by sending SMS messages. The design of the CTMC system was inspired by the Upoc (`http://www.upoc.com/`) and Dodgeball (`http://www.dodgeball.com/`) services.

A typical CTMC use case is as the follows. Harry enters a smart meeting room. The Context Broker detects Harry's presentspresence. Knowing he is the host of the meeting, the Context Broker infers the kind of services that are relevant to the tasks of a meeting host and sends the corresponding CTMC commands of these services to Harry's cellphone. This set of commands includes presentation set-up controls, room temperature and lighting condition controls, and meeting scheduler controls. Feeling kind of warm in the room, Harry adjusts the room temperature by sending the appropriate CTMC command to the temperature service. Wondering if any of the participants are running late for the meeting, Harry sends a notification request command to the meeting scheduler.

In the current prototype implementation, I have developed CTMC commands for searching the talk events that are scheduled at the UMBC eBiquity group meetings and the commands for requesting follow-up resources of the talks (e.g., presentation slides, white papers). Talk announcements and follow-up resources are typically published on the eBiquity web site in both HTML and OWL representations. The Context Broker monitors these publish ontology documents and uses this information to answer the incoming CTMC search queries and requests.

The list of the supported CTMC commands:

`qMEETING` the user asks the Context Broker to list the title and the ID of today's meeting events. If no event is scheduled today, then the Broker answers no meeting event found.

`qSPEAKER [meeting-ID]` the user asks the Context Broker to send a short description of the meeting speaker, which includes the speaker's name, associated institution, and professional title, the number of publications that the speaker has made, and the names of the research projects that the speaker is currently involved in.

`qINFO [meeting-ID]` the user asks the Context Broker to send time, location, and status information about a meeting.

`qFOLLOWUP [meeting-ID] [email-address]` the user requests the Context Broker to forward documents and resources that are associated with the meeting to a designated email address. When the command is received, the Context Broker sends a brief event description, the number of people who attended the meeting, and any posted PowerPoint presentations to the designated email address.

In the current implementation, when a user sends a CTMC command via SMS, the message is relayed to an email address of the Context Broker through an intermediate SMS routing service. This service is provided through an existing cellular network provider. Periodically, the Context Broker checks the designated email account for any incoming CTMC commands. Once a command is received and processed, the result is sent back to the user's cellphone via the SMS routing service. Figure 22 shows a UML diagram of this process.

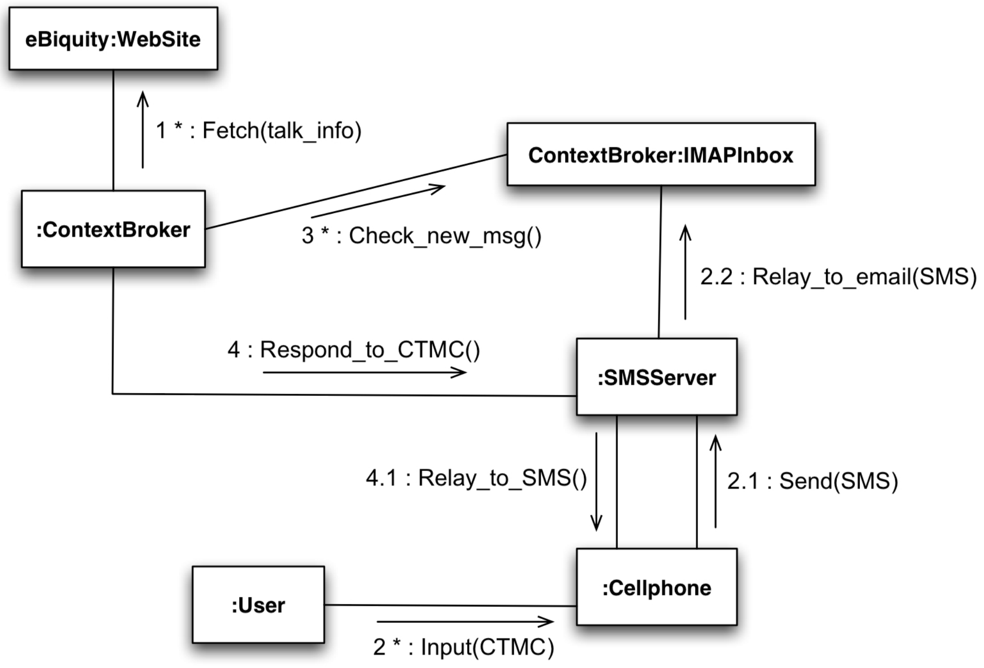


Figure 22: A UML collaboration diagram that shows an instant of the CTMC execution process.

Chapter VIII.

CONCLUSIONS

Context-awareness is an important aspect of the ubiquitous computing vision. It enables computing systems to provide relevant services and information based on the situational needs of the users. Through an understanding of context, computing systems will no longer require the users to give explicit instructions at every step during their interaction with the systems. By exploiting context, context-aware systems will weave themselves into the background of our everyday life.

I believe the intelligence of context-aware systems will be limited if the systems are unable to effectively represent and reason about contexts, and users will abandon the most useful context-aware services if they are unable control the sharing of their private information. I developed CoBrA, a new broker-centric agent architecture for supporting pervasive context-aware systems. CoBrA differs from other similar systems in using ontologies expressed in the OWL language for context representation and modeling, rule-based logical inference for context reasoning, and declarative policies for privacy protection.

From the prototype systems that I have implemented, I showed CoBrA is a feasible architecture to support context-aware systems. In particular, CoBrA helped to reduce the time and effort to rapidly prototype the EasyMeeting system. The use of SOUPA and CoBrA ontologies helped context-aware services to represent context and share knowledge. Using the CoBrA Demo Toolkit, I successfully demonstrated a policy-based approach for privacy protection.

A. Lessons Learned

CoBrA can help developers to build context-aware systems. My research showed that (i) the use of CoBrA can reduce the effort and time to rapidly prototype context-aware applications, (ii) ontologies expressed using the OWL language can provide a uniformed solution for context representation and reasoning, knowledge sharing, and meta-language definitions, and (iii) the use of logical inference can improve the reasoning capability of a context-aware system.

1. CoBrA Improves Rapid Prototyping

Prior to CoBrA, defining context representations and context reasoning procedures are the necessary tasks of building a new context-aware system. These tasks are often time consuming because context modeling is an iterative process that requires the special knowledge of a domain expert [36]. The greater the number of systems that need to be implemented, the more time and effort would be required to develop the context representation and the reasoning procedures.

I showed that CoBrA can improve rapid prototyping by reducing the time and effort required to define context representations and context reasoning procedures. In EasyMeeting, different context-aware services use different contextual information and require different context reasoning support. Because CoBrA provides a set of common ontologies, the development of these services can reuse the existing representation of context. As the context broker plays the role in maintaining a shared model of context, the implementation of different context-aware services can reply on the context broker to provide the needed contextual information. In addition, the context reasoning of these services also can be simplified because the context broker provides the logical inference to detect and resolve inconsistent information.

2. Context-Aware Systems Needs Ontology

The use of ontologies expressed in the OWL langauge distinguishes CoBrA from other infrastructures that are also aimed to support context-aware systems [21]. In the past, the use of programming data structures (e.g., Java classes) to model context often tightly couples the context representation with the system implementation. The resulting system often has a rigid implementation and is difficult to maintain. Because context models that are represented with programming objects lack declarative semantics, they cannot effectively support knowledge sharing and context reasoning. In CoBrA, these issues are addressed by exploiting the use of ontologies.

I showed that ontologies expressed using the OWL language can provide a uniform solution for context representation and reasoning, knowledge sharing, and policy definition. Using OWL, various types of contextual information can be expressed as ontologies with declarative semantics. By sharing a common ontology, independently developed context-aware systems are enabled to communicate and share knowledge.

The OWL langauge also provides a specification for building context reasoners (e.g., the context broker's temporal, spatial, and policy reasoners). Because the reasoners are built on a standard specification, they can be reused by different systems to support context reasoning.

I demonstrated the use of the SOUPA policy ontology for privacy protection. It showed that the OWL language is suitable for defining new special purpose meta-languages. In the past, meta-languages typically do not share a common root of representation (e.g., the Ponder policy language has a procedure-oriented representation [35], and the Web Service Description Language has an XML-based representation [27]). Without a shared representation, it is difficult to integrate the functions of different meta-languages in a single system. The OWL language can help address this issue. From the prototype implementations, I showed that the OWL language can be used to express a policy language and its associated policy reasoning can integrate with the ontology vocabularies that are developed for other domain purposes (e.g., ontologies for describing people's profile and location context).

3. Logical Inference Helps to Enable Context-Awareness

In order to understand context, computing systems must be given the ability to interpret sensing information and maintain a consistent model of this information. Logical inference is a means to enable context-awareness. Through reasoning, the computing systems can derive conclusions about the acquired contextual information and find evidences to argue against inconsistent information that is acquired from the external sources.

CoBrA showed the use of rule-based logical inference to enable context-awareness. In particular, it demonstrated the use of ontology reasoning and domain heuristics reasoning to support context interpretation. I showed that a rule-based implementation of context reasoning allows a context-aware system to exploit logical inference features (e.g., assumption-based reasoning, argumentation) that were not directly available in the procedure-based implementations.

The ability to detect and resolve inconsistent information is of great importance to a context-aware system.

Using the Theorist system, I showed the assumption-based reasoning technique can be used to help a context broker to resolve inconsistent information. Because the assumptions used to interpret context are represented in logical rules, an argumentation behavior can be implemented to enable a team of context brokers to resolve inconsistent information through dialogs.

B. Future Works

The CoBrA prototypes demonstrated the reasoning and knowledge sharing features of a context broker. However, the current context broker implementation can provide and process only contextual information that is associated the meeting domain. Part of the future works is to increase the breadth of a context broker's knowledge and expand its context reasoning capability.

One way to increase a context broker's knowledge is to broaden its source of context. The more information that the context broker can acquire and reason with, the more context knowledge that it can derive. To demonstrate this idea, I propose a new CoBrA sensing infrastructure be developed. The goal is to create a flexible middleware system to facilitate the dynamic discovery of context sensors and the invocation of sensing procedures.

This new infrastructure will have the following key components: (i) context sensing ontology, (ii) sensing service registry, and (iii) service composition manager. The context sensing ontology is a set of ontologies for describing the sensor profiles (e.g., the sensor type, device interfaces) and capabilities (e.g., the kind of contextual information that the sensor can provide, the scores that estimate the accuracy of acquired sensing data), and the service interface of the sensors (i.e., ontologies that describes the usage of the sensors). This ontology is used by a context broker to discover sensors and to interact with their sensing interfaces. The sensing service registry is a directory service for publishing and discovering sensing services. The actual implementation of this service registry may utilize some of the existing service discovery architecture (e.g., UDDI [78] or Jini [102]). The service composition manager is a specialized agent that facilities sensing service composition based on the needs of the context broker. When the context broker is unable to discover a single service that would supply a desired type of contextual information, it sends a service composition request to the service composition manager. The manger will attempt to fulfill the request through service composition.

As the source of context increases, the context broker will be forced to reason with and maintain additional amounts of contextual knowledge. In theory, the existing rule-based implementation could be extended to support the increasing context reasoning tasks. However, the rule-based implementation has certain weaknesses. First, it might not scale well when the number of conditional parameters increases in the logical inference rules. Second, because building context reasoning rules often requires the special knowledge of domain experts, it might be difficult to effectively define reasoning rules for all possible contexts that a system will use.

To address this issue, part of the future works is to explore alternative approaches for context reasoning. I believe probabilistic approaches could be used to supplement the weaknesses of the rule-based approach. For example, the context broker's implementation could exploit the use of a two-state hidden Markov model (HMM) for inferring the location of a person and an indicator of whether or not a person is in motion [65]. Krumm and Horvitz [65] show that their probabilistic approach can be used to infer over the location data collected from an indoor Wi-Fi network. In their experiments, the probabilistic reasoner is able to determine the location of a person with a median error of 1.5 meters and decide whether or not a person is in motion with a classification accuracy of 87%.

I believe using a probabilistic approach to infer context will add certain flexibility to the implementation. Unlike the rule-based systems, probabilistic systems are typically more adaptive to the changes of the environment. In an open and dynamic environment, the developers of a smart space cannot always anticipate all possible changes that could occur in the space. By adding probabilistic reasoning into CoBrA, we give developers more flexility in choosing to use different types of context reasoning.

Privacy protection is part of a context broker's responsibility. Before sharing a user's information, the context broker consults the privacy policy to see if the client agent is permitted to acquire this information. The information is shared only if the policy grants the permission. While the present approach can effectively enforce policy on the context broker, however, it cannot enforce policy on the client agent once the user's information has been shared.

A possible solution to this problem is to introduce a reputation ranking system. This system will monitor the knowledge sharing behavior of different agents. Based on their compliance with the user-defined policies, the system will increase or decrease the reputation value of the agents. A simple ranking strategy might be that whenever an agent shares information, the system increases the agent's reputation value if no privacy policy rule has been violated, and decrease the value if one or more policy rules have been violated. An actual implementation of this system may include a trust evaluation model [56] for building reputation ranking based on different types of trust (e.g., trust that results from role-based relationships between two agents and trust that ensues from the direct interactions between two agents).

BIBLIOGRAPHY

[1] Gregory D. Abowd, Anind K. Dey, Robert Orr, and Jason A. Brotherton. Context-awareness in wearable and ubiquitous computing. In *ISWC*, pages 179–180, 1997.

[2] Philip E. Agre. Changing places: Contexts of awareness in computing. *Special Issue on Context-Aware Computing. Human-Computer Interaction*, 16(2-4), 2001.

[3] Jalal Al-Muhtadi, Roy Campbell, Apu Kapadia, Dennis Mickunas, and Seung Yi. Routing through the mist: Privacy preserving communication in ubiquitous computing environments. In *Proceedings of the International Conference of Distributed Computing Systems (ICDCS 2002)*, Vienna, Austria, 2002.

[4] Abhaya Asthana, Mark Cravatts, and Paul Krzyzanowski. An indoor wireless system for personalized shopping assistance. In *IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, US, 1994.

[5] Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. *OWL Web Ontology Language Reference*, w3c recommendation 10 february 2004 edition, February 2004.

[6] Fabio Bellifemine, Giovanni Caire, and Tiziana Trucco. *JADE Programmer's Guide*. TILAB and University of Parma, 2004.

[7] Victoria Bellotti and Abigail Sellen. Design for privacy in ubiquitous computing environments. In *Proceedings of the Third European Conference on Computer Supported Cooperative Work (ECSCW'93)*, pages 77–92. Kluwer, 1993.

[8] Massimo Benerecetti, Paolo Bouquet, and Matteo Bonifacio. Distributed context-aware systems. *Human Computer Interaction. Special issue on Context-aware computing*, 16:213–228, 2001.

[9] Frazer Bennett, Tristan Richardson, and Andy Harter. Teleporting - Making Applications Mobile. In *Proceedings of 1994 Workshop on Mobile Computing Systems and Applications*, Santa Cruz, December 1994.

[10] Alastair R. Beresford and Frank Stajano. Location privacy in pervasive computing. *IEEE Pervasive computing*, 2(1), January–March 2003.

[11] Tim Berners-Lee. *Primer: Getting into RDF & Semantic Web using N3*, 2003.

[12] Paul V. Biron and Ashok Malhotra. *XML Schema Part 2: Datatypes*, 2001.

[13] Jeffrey M. Bradshaw, Andrzej Uszok, Renia Jeffers, Niranjan Suri, Patrick J. Hayes, Mark H. Burstein, A. Acquisti, Brett Benyo, Maggie R. Breedy, Marco M. Carvalho, David J. Diller, Matt Johnson, Shriniwas Kulkarni, James Lott, Maarten Sierhuis, and Ron Van Hoof. Representation and reasoning about daml-based policy and domain services in kaos. *Proceedings of The Second International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS2003)*, 2003.

[14] Dan Brickley and R.V. Guha. Rdf vocabulary description language 1.0: Rdf schema. In *W3C Recommendation*. RDF Core Working Group, 2004.

[15] Dan Brickley and Libby Miller. *FOAF Vocabulary Specification*, revision 1.47 edition, Sept 2003.

[16] Roy Campbell, Jalal Al-Muhtadi, Prasad Naldurg, Geetanjali Sampemane1, and M. Dennis Mickunas. Towards security and privacy for pervasive computing. In *Proceedings of International Symposium on Software Security*, Tokyo, Japan, 2002.

[17] Licia Capra, Wolfgang Emmerich, and Cecilia Mascolo. Reflective middleware solutions for context-aware applications. *Lecture Notes in Computer Science*, 2192, 2001.

[18] Jeremy Carroll, Ian Dickinson, Chris Dollin, Dave Reynolds, Andy Seaborne, and Kevin Wilkinson. Jena: Implementing the semantic web recommendations. Technical Report HPL-2003-146, Hewlett Packard Laboratories, 2003.

[19] Paul Castro and Richard Muntz. Using context to assist in multimedia object retrieval applications. In *Proceedings of the ACM Workshop on Multimedia Intelligent Storage and Retrieval Management*, 1999.

[20] Guanling Chen and David Kotz. A survey of context-aware mobile computing research. Technical Report TR2000-381, Dartmouth College, Computer Science, Hanover, NH, Nov 2000.

[21] Harry Chen, Tim Finin, and Anupam Joshi. An ontology for context-aware pervasive computing environments. *Special Issue on Ontologies for Distributed Systems, Knowledge Engineering Review*, 18(3):197–207, 2004.

[22] Harry Chen, Tim Finin, and Anupam Joshi. Semantic web in in the context broker architecture. In *Proceedings of PerCom 2004*, March 2004.

[23] Harry Chen, Filip Perich, Dipanjan Chakraborty, Tim Finin, and Anupam Joshi. Intelligent agents meet semantic web in a smart meeting room. In *Proceedings of the Thrid International Joint Conference on Autonomous Agents & Multi-Agent Systems*, July 2004.

[24] Harry Chen, Filip Perich, Tim Finin, and Anupam Joshi. SOUPA: Standard ontology for ubiquitous and pervasive applications. In *Proceedings of the First International Conference on Mobile and Ubiquitous Systems: Networking and Services*, 2004.

[25] Harry Chen and Sovrin Tolia. Steps towards creating a context-aware agent system. Technical report, Hewlett Packard Labs, 2001.

[26] Amit Choudhri, Lalana Kagal, Anupam Joshi, Tim Finin, and Yelena Yesha. PatientService : Electronic Patient Record Redaction and Delivery in Pervasive Environments. In *Fifth International Workshop on Enterprise Networking and Computing in Healthcare Industry (Healthcom 2003)*, June 2003.

[27] Erik Christense, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. *Web Services Description Language (WSDL) 1.1 (W3C Note 15 March 2001)*. W3C, 2001.

[28] Michael H. Coen. Building brains for rooms: Designing distributed software agents. In *Proceedings of Ninth Conference on Innovative Applications of Artificial Intelligence*, pages 971–977, 1997.

[29] Michael H. Coen. Design principles for intelligent environments. In *Proceedings of AAAI/IAAI 1998*, pages 547–554, 1998.

[30] Philip Cohen, Hector Levesque, and Ira Smith. On team formation. In Ghita Holmstrm-Hintikka and Raimo Tuomela, editors, *Comteporary Action Theory*, volume 2. Kluwer Academic Publishers, 1997.

[31] Dan Connolly, Frank van Harmelen, Ian Horrocks, Deb McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. *DAML+OIL Reference Description*, march 2001 edition, 2001.

[32] Simon Cox, Paul Daisey, Ron Lake, Clemens Portele, and Arliss Whiteside. Geography markup language (gml 3.0). In *OpenGIS Documents*. OpenGIS Consortium, 2003.

[33] Lorrie Cranor, Marc Langheinrich, Massimo Marchiori, Martin Presler-Marshall, and Joseph Reagle. *The Platform for Privacy Preferences 1.0 (P3P1.0) Specification*, jan 2002.

[34] Nadine Cullot, Christine Parent, Stefano Spaccapietra, and Christelle Vangenot. Ontologies : A contribution to the DL/DB debate. In *Proceedings of the First International Workshop on Semantic Web and Databases*, 2003.

[35] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The ponder policy specification language. *Lecture Notes in Computer Science*, 1995:18–??, 2001.

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

[37] P. Dourish. Seeking a foundation for context-aware computing. *Human-Computer Interaction*, 2001. `http://citeseer.nj.nec.com/dourish01seeking.html`.

[38] Edd Dumbill. Finding friends with xml and rdf. In *IBM developerWorks, XML Watch*. xmlhack.com, June 2002.

[39] Thomas Erickson. Some problems with the notation of context-aware computing. *Communications of the ACM*, pages 102–104, 2004.

[40] Irfan A. Essa. Ubiquitous sensing for smart and aware environments. *IEEE Personal Communications*, pages 46–49, October 2000.

[41] Tim Finin, Yannis Labrou, and James Mayfield. KQML as an agent communication language. *Software Agents*, pages 291–316, 1997.

[42] Foundation for Intelligent Physical Agent. *FIPA Device Ontology Specification*, pc00091a edition, 2001.

[43] Foundation for Intelligent Physical Agents. *FIPA Query Interaction Protocol Specification*, 2002.

[44] The Foundations for Intelligent Physical Agents. *FIPA Abstract Architecture Specification*, sc00001l edition, December 2002.

[45] Ernest J. Friedman-Hill. *Jess, The Expert System Shell for the Java Platform*. Sandia National Laboratories, version 6.1a4 edition.

[46] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, 1995.

[47] Fabien L. Gandon and Norman M. Sadeh. Semantic web technologies to reconcile privacy and context awareness. *Web Semantics Journal*, 1(3), 2004.

[48] Volker Haarslev and Ralf Möller. Racer system description. In *Proceedings of the International Joint Conference on Automated Reasoning 2001*, 2001.

[49] Nicholas Hanssens, Ajay Kulkarni, Rattapoom Tuchinda, and Tyler Horton. Building agent-based intelligent workspaces. In *Proceedings of ABA Conference*, June 2002. To Appear.

[50] Albert Held, Sven Buchholz, and Alexander Schill. Modeling of context information for pervasive computing applications. In *Proceedings of the 6th World Multiconference on Systemics, Cybernetics and Informatics (SCI2002)*, 2002.

[51] Ken Hinckley, Jeffrey S. Pierce, Mike Sinclair, and Eric Horvitz. Sensing techniques for mobile interaction. In *UIST*, pages 91–100, 2000.

[52] Jerry R. Hobbs. A daml ontology of time. `http://www.cs.rochester.edu/~ferguson/daml/daml-time-20020830.txt`, 2002.

[53] Ian Horrocks. FaCT and iFaCT. *Description Logics*, 1999.

[54] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. *SWRL: A Semantic Web Rule Language – Combining OWL and RuleML*. W3C, 2004.

[55] HP Labs 2. *HOWTO Use Jena and DIG Reasoners*, 2004.

[56] Dong Huynh, Nicholas R. Jennings, and Nigel R. Shadbolt. Developing an integrated trust and reputation model for open multi-agent systems. In *Proceedings of the 7th Workshop on Trust, Privacy, Deception and Fraud In Agent Societies*, 2004.

[57] Lalana Kagal, Tim Finin, and Anupam Joshi. A Policy Based Approach to Security for the Semantic Web. In *2nd International Semantic Web Conference (ISWC2003)*, September 2003.

[58] Lalana Kagal, Tim Finin, and Anupam Joshi. A policy language for a pervasive computing environment. In *IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, 2003.

[59] Lalana Kagal, Vlad Korolev, Harry Chen, Anupam Joshi, and Timothy Finin. Centaurus : A framework for intelligent services in a mobile environment. In *Proceedings of the International Workshop on Smart Appliances and Wearable Computing*, 2001.

[60] Lalana Kagal, Massimo Paolucci, Naveen Srinivasan, Grit Denker, Tim Finin, and Katia Sycara. Authorization and privacy for semantic web services. *AAAI 2004 Spring Symposium on Semantic Web Services*, March 2004.

[61] Lalana Kagal, Massimo Paoucci, Naveen Srinivasan, Grit Denker, Tim Finin, and Katia Sycara. Authorization and privacy for semantic web services. *IEEE Intelligent Systems (Special Issue on Semantic Web Services)*, 19(4):50–56, July 2004.

[62] Lalana Kagal, James Parker, Harry Chen, Anupam Joshi, and Tim Finin. *Handbook of Mobile Computing*, chapter Security, Trust and Privacy in Mobile Computing Environments. CRC Press, 2004.

[63] Tim Kindberg and John Barton. A web-based nomadic computing system. *Computer Networks*, 35(4):443–456, 2001.

[64] Graham Klyne, Franklin Reynolds, Chris Woodrow, Hidetaka Ohto, Johan Hjelm, Mark H. Butler, and Luu Tran. *Composite Capability/Preference Profiles (CC/PP): Structure and Vocabularies (W3C Working Draft 25 March 2003)*. W3C, 2003.

[65] John Krumm and Eric Horvitz. LOCADIO: Inferring motion and location from wi-fi signal strengths. In *Proceedings of the First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services*, 2004.

[66] Sanjeev Kumar, Philip R. Cohen, and Hector J. Levesque. The adaptive agent architecture: Achieving fault-tolerance using persistent broker teams. In *Proceedings of the Fourth International Conference on Multi-Agent Systems*, pages 159–166, 2000.

[67] Mik Lamming and Mike Flynn. Forget-me-not: intimate computing in support of human memory. In *Proceedings FRIEND21 Symposium on Next Generation Human Interfaces*, 1994.

[68] Marc Langheinrich. Privacy by design – principles of privacy-aware ubiquitous systems. In G.D. Abowd, B. Brumitt, and S. Shafer, editors, *Proceedings of Ubicomp 2001*, volume 2201 of *Lecture Notes in Computer Science*, pages 273–291. Springer, 2001.

[69] Marc Langheinrich. A privacy awareness system for ubiquitous computing. In *Proceedings of Ubi-Comp 2002*, 2002.

[70] Ora Lassila and Ralph R. Swick. Resource description framework (rdf) model and syntax specification. www.w3c.org, feb 1999.

[71] Douglas B. Lenat and R. V. Guha. *Building Large Knowledge-Based Systems: Representation and Inference in the Cyc Project*. Addison-Wesley, February 1990.

[72] Gary Look and Howard Shrobe. A plan-based mission control center for autonomous vehicles. In *Proceedings of the 9th International Conference on Intelligent User Interfaces*, pages 277–279. ACM Press, 2004.

[73] Peter McBurney and Simon Parsons. Locutions for argumentation in agent interaction protocols. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, volume 3, 2004.

[74] John McCarthy and Sasa Buvac. Formalizing context (expanded notes). In Sasa Buvač and Łucia Iwańska, editors, *Working Papers of the AAAI Fall Symposium on Context in Knowledge Representation and Natural Language*, pages 99–135, Menlo Park, California, 1997. American Association for Artificial Intelligence.

[75] Deborah L. McGuinness and Frank van Harmelen. *OWL Web Ontology Language Overview*. W3C, 2003. `http://www.w3.org/TR/owl-features/`.

[76] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. In *Sixteen ACM Symposium on Operating Systems Principles*, pages 276–287, Saint Malo, France, 1997.

[77] Natalya Fridman Noy and Deborah L. McGuinness. Ontology development 101: A guide to creating your first ontology. Technical Report KSL-01-05, Stanford Knowledge Systems Laboratory, 2001.

[78] Organization for the Advancement of Structured Information Standards. *Introduction to UDDI: Important Features and Functional Concepts*, 2004.

[79] Feng Pan and Jerry R. Hobbs. Time in owl-s. In *Proceedings of AAAI-04 Spring Symposium on Semantic Web Services*, Stanford University, California, 2004.

[80] Peter F. Patel-Schneider, Patrick Hayes, and Ian Horrocks. *OWL Web Ontology Language Semantics and Abstract Syntax (W3C Recommendation 10 February 2004)*. W3C, 2004.

[81] Filip Perich. A service for aggregating and interpreting contextual information. Technical report, Hewlett Packard Labs, 2002.

[82] Filip Perich. *MoGATU BDI Ontology*, 2004.

[83] Stephen Peters and Howie Shrobe. Using semantic networks for knowledge representation in an intelligent environment. In *1st Annual IEEE International Conference on Pervasive Computing and Proceedings of the 1st Annual IEEE International Conference on Pervasive Computing and Communications (PerCom'03)*, March 2003.

[84] Sadie Plant. On the mobile: the effects of mobile telephones on social and individual life. Technical report, Motorola, 2000.

[85] David Poole. Explanation and prediction: an architecture for default and abductive reasoning. *Computational Intelligence*, 5(2):97–110, 1989.

[86] David Poole. *Local Users Guide to Theorist*. University of British Columbia, 1990.

[87] David Poole. Compiling a default reasoning system into prolog. *New Generation Computing*, 9(1):3–38, 1991.

[88] Shelley Powers. *Practical RDF*. O'Reilly & Associates, 2003.

[89] Nissanka B. Priyantha, Anit Chakraborty, and Hari Balakrishnan. The cricket location-support system. In *Mobile Computing and Networking*, pages 32–43, 2000.

[90] David A. Randell, Zhan Cui, and Anthony G. Cohn. A spatial logic based on regions and connection. In *Proceedings of the 3rd International Conference on Knowledge Representation and Reasoning*, 1992.

[91] Jun Rekimoto. Tilting operations for small screen interfaces. In *ACM Symposium on User Interface Software and Technology*, pages 167–168, 1996.

[92] Dave Reynolds. *Jena 2 Inference Support*, April 2004.

[93] Daniel Salber, Anind K. Dey, and Gregory D. Abowd. The context toolkit: Aiding the development of context-enabled applications. In *Proceedings of CHI'99*, pages 434–441, 1999.

[94] Bill Schilit, Norman Adams, and Roy Want. Context-aware computing applications. In *IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, US, 1994.

[95] William Noah Schilit. *A System Architecture for Context-Aware Mobile Computing*. PhD thesis, Columbia University, 1995.

[96] A. Schmidt, K. A. Aidoo, A. Takaluoma, U. Tuomela, K. Van Laerhoven, and W. Van de Velde. Advanced interaction in context. *Lecture Notes in Computer Science*, 1707:89–??, 1999.

[97] Albrecht Schmidt, Michael Beigl, and Hans-W. Gellersen. There is more to context than location. *Computers & Graphics Journal*, 23(6):893–902, December 1999.

[98] Andy Seaborne. *RDQL - A Query Language for RDF (W3C Member Submission 9 January 2004)*. W3C, 2004.

[99] B.I.J. Siljee, I.E. Bosloper, and J.A.G. Nijhuis. A classification framework for storage and retrieval of context. In *Proceedings of KI-04 Workshop on Modeling and Retrieval of Context*, 2004.

[100] Morris Sloman and Emil Lupu. Security and management policy specification. *IEEE Network, Special Issue on Policy-Based Networking*, 2002.

[101] Michael K. Smith, Chris Welty, and Deborah McGuinness. Owl web ontology language guide. `http://www.w3.org/TR/owl-guide/`, 2003.

[102] Sun Microsystems. *Jini Architectural Overview: Technical White Paper*, 1999.

[103] Milind Tambe, Paul Scerri, and David V. Pynadath. Adjustable autonomy for the real world. In *Proceedings of AAAI Spring Symposium on Safe Learning Agents 2002*, 2002.

[104] Gianluca Tonti, Jeffrey M. Bradshaw, Renia Jeffers, Rebecca Montanari, Niranjan Suri, and Andrzej Uszok. Semantic web languages for policy representation and reasoning: A comparison of kaos, rei, and ponder. In *Proceedings of the 2nd International Semantic Web Conference (ISWC2003)*, 2003.

[105] Roy M. Turner. Context-sensitive reasoning for autonomous agents and cooperative distributed problem solving. In *Proceedings of the IJCAI Workshop on Using Knowledge in its Context*, Chambéry, France, 1993.

[106] Jeffrey Undercoffer, Filip Perich, Andrej Cedilnik, Lalana Kagal, Anupam Joshi, and Tim Finin. A secure infrastructure for service discovery and management in pervasive computing. *The Journal of Special Issues on Mobility of Systems, Users, Data and Computing*, 2003.

[107] Roy Want, Andy Hopper, Veronica Falcao, and Jon Gibbons. The active badge location system. Technical Report 92.1, Olivetti Research Ltd., ORL, 24a Trumpington Street, Cambridge CB2 1QA, 1992.

[108] Marc Weiser. The computer for the 21st century. *Scientific American*, 265(30):94–104, 1991.

[109] Michael J. Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2):115–152, June 1995.

[110] Huadong Wu, Mel Siegel, and Sevim Ablay. Sensor fusion for context-understanding. In *Proceedings of IEEE Instrumentation and Measurement Technology Conference 2002*, 2002.