Hyperglue: An Infrastructure for Human-Centered Computing in Distributed, Pervasive, Intelligent Environments

by

Stephen L. Peters

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2006

Author .... [illegible] ....
Department of Electrical Engineering and Computer Science
February 3, 2006

Certified by .... [illegible] ....
Howard E. Shrobe
Principal Research Scientist, MIT CSAIL
Thesis Supervisor

Accepted by .... [illegible] ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Hyperglue: An Infrastructure for Human-Centered Computing in Distributed, Pervasive, Intelligent Environments

by

Stephen L. Peters

Submitted to the Department of Electrical Engineering and Computer Science
on February 3, 2006, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

Abstract

As intelligent environments (IEs) move from simple kiosks and meeting rooms into the everyday offices, kitchens, and living spaces we use, the need for these spaces to communicate not only with users, but also with each other, will become increasingly important. Users will want to be able to shift their work environment between localities easily, and will also need to communicate with others as they move about. These IEs will thus require two pieces of infrastructure: a knowledge representation (KR) which can keep track of people and their relationships to the world; and a communication mechanism so that the IE can mediate interactions.

This thesis seeks to define, explore and evaluate one way of creating this infrastructure, by creating societies of agents that can act on behalf of real-world entities such as users, physical spaces, or informal groups of people. Just as users interact with each other and with objects in their physical location, the agent societies interact with each other along communication channels organized along these same relationships. By organizing the infrastructure through analogies to the real world, we hope to achieve a simpler conceptual model for the users, as well as a communication hierarchy which can be realized efficiently.

Thesis Supervisor: Howard E. Shrobe
Title: Principal Research Scientist, MIT CSAIL

Acknowledgments

Much thanks must go to the entire staff of Project AIRE at MIT CSAIL, who over the past years have provided grand service to this development and research, acting as sounding boards, test beds, coders, and in general being great people. I'd like to particularly recognize Luke Weisman, Krzysztof Gajos, Kevin Quigley, Gary Look, Max van Kleek, Tyler Horton, and Michael Coen.

Howard Shrobe, leader of Project AIRE and mentor to all, was absolutely invaluable in serving as a suggestion box and assisting in refining several elements of the system design.

Finally, I dedicate this work to my wife Mary Linton, who has stood by my side from Cambridge to Kyang Tung, and everywhere in between.

Contents

List of Figures

Chapter 1

Claim

1.1 Background

The field of pervasive computing has expanded rapidly over the past decade, since Mark Weiser first began to articulate a vision of technology that will be incorporated seamlessly into the world at large [44]. Weiser's descriptions of these technologies suggested a move to a future where computation is embedded in many of the everyday components that surround us in our daily lives, from active Post-It notes to large wall-mounted intelligent blackboards. Over time, the notion of taking a space and equipping it with this kind of technology has become known as pervasive or ubiquitous computing ("ubicomp"), and has driven research into how this technology can be used to assist people going about everyday tasks.

When implementing technologies to accommodate this vision, researchers often imagine implementations that can span and unite a large spectrum of appliances, ranging from small personal handheld devices (such as personal organizers or cellphones) to rooms or buildings that can recognize and respond to their inhabitants. As such, we need technologies that can collect and incorporate the information provided by such disparate hardware, and allow the users to interact with the information and devices available to them. Ideally, they will accomplish this with a minimum of intrusion into the user's activities.

In the space of these ubiquitous computing applications, a common subfield is that of IEs, focusing on the implications of humans interacting with highly-instrumented spaces. These spaces can include business offices, conference rooms, laboratories, domestic living rooms, and even kitchens. The sensors used to retrieve information about these environments are also varied, employing cameras, microphones, radio-frequency identifiers [39], floor-mounted pressure sensors [14], or simple detectors within the many hinges and switches such environments contain. To provide a reactive component to the space, the computing systems behind the IE often have control over devices in the room, such as lights, thermostats, speakers, computer displays, or even more advanced devices that can display information on any surface in the room [35]. They may even make use of a person's portable devices, such as PDAs, laptops, or cell phones. Although the spaces and devices change, the research focus has a common thrust – the sensors are used to detect the activities and desires of the human occupants, and the devices are used to provide the humans with information they seek, or to adjust the environment to their needs.

Because these systems are designed to be augmented living spaces, they often seek to assist with human-human communication as well as with the direct human-environment interactions. However, such communication also needs to recognize that the spaces we live and work in often serve as private refuges, and thus must temper the desire to facilitate communication with the world with the individual's needs for privacy and peace. As such, many IEs utilize agent systems that operate on behalf of the users, to mediate and moderate the communication, or even respond on behalf of their user without interrupting them.

Most research in the field of intelligent environments (IEs) confines itself to exploring the infrastructure for a single environment, and pays scant attention to exploring the possibilities and requirements of a multitude of IEs acting in concert and in communication with each other. Thus multiple-IE focus becomes even more essential as the technology that enables such environments moves from contained spaces such as offices and laboratories and into more group-focused locales such as conference rooms or common areas, and even onto personal handheld devices.

This thesis describes an implemented infrastructure for IEs that can support hundreds of people, and interoperate with other environments to support the user's needs.

Imagine, if you will, an administrative assistant for a research group, wishing to announce to the local graduate student population that some food that was left over from a presentation is now being made available in a nearby lounge.

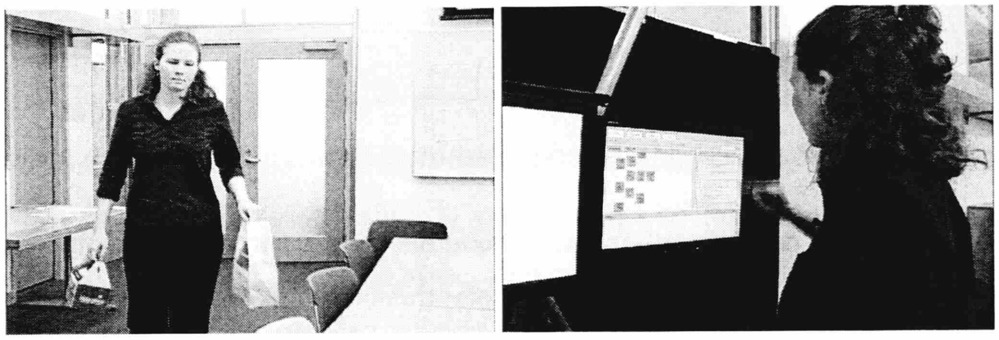
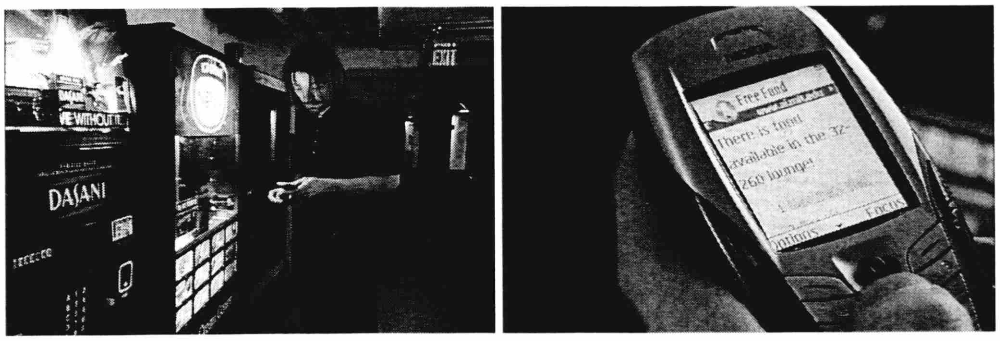


Figure 1-1: Ellie using a kiosk to contact graduate students.

Ellie carries the food into the common area, places it in view of the lounge's cameras, and then turns to a nearby kiosk to send the message. After turning on the kiosk's microphone, she asks the kiosk to "please tell the graduate students that food is available in the lounge." The kiosk briefly informs her of how many people the message will reach, and Ellie continues on with her day.



There is food available in the 32-260 lounge!

Figure 1-2: Max receives the message on a cell phone.

About this time, Max is attempting to grab a quick snack from a vending machine near to the lab. As he stares at the wealth of choices available to him, his phone vibrates with the message from Ellie, informing him about the leftovers and thus potentially rescuing him from a lunch of overpriced chips.

Figure 1-3: Using a cellphone to explore additional information about a message.

Max starts up an assistant application on his phone, through which he can ask questions like "how many people have seen this message," to decide whether it's worth pursuing this particular quarry. He also asks the system to "show me a picture of the lounge," which then requests that the kiosk's webcam snap a picture of the room and send it to him on his handheld. Max is both intrigued and starving, so he hurriedly makes his way towards the lounge before too many other graduate students receive the message and get a similar notion.

Although there are certainly research questions to be raised at the levels of user interface in this scenario (e.g, speech parsing and understanding, as well as interface design for the kiosk and cellphones), for this thesis I focus on the machinery necessary for the underlying infrastructure to perform the message delivery.

- Ellie simply says "tell the graduate students" about the food in the lounge. She clearly doesn't mean all the graduate students in the universe, so there's an implication that she's referring to a subset of them – in this case, the graduate students who are associated with the lounge, perhaps only members of research groups sharing that space.

- Ellie doesn't bother specifying which lounge she's currently located in – the kiosk she is using has a clear notion of where it is located and therefore which lounge is meant. However, even if Ellie were using a handheld like Max's, there should be no change in her interaction; the handheld should also have knowledge of its current location and provide that information automatically.

- The message really *shouldn't* be delivered to every possible graduate student associated with the lounge; instead it should be limited to those (like Max) who are near enough to act on the message in a reasonable time frame. For example, sending a message about free food to a grad student currently attending a conference thousands of miles away is useless.

- This particular message is effectively time-limited; even when the student is nearby, if he won't see the message for three hours, the food will be gone and the information wasted. Thus, it would be nice to deliver this information as swiftly as possible, and the system should be able to use whatever tools are at hand to do that. If Max were at his desk, it might be faster to pop up the message on his display.

- Despite the time-sensitive nature of this message, it isn't really an urgent message, and does not have to be delivered to every recipient as quickly as possible, and shouldn't intrude in situations where the recipient is otherwise occupied. For example, using a conference room display in an ongoing meeting in order to announce to a meeting attendee that there is food nearby would be poor "ettiquette" for an IE. However, there are times where intrusive behavior would be absolutely appropriate – if Ellie were announcing an emergency in the building, it should be delivered to all the recipients as quickly as possible.

- Users should be able to receive this information any way they like. Max may prefer to get messages on his cellphone, even if he's seated at his desk. Others may have different preferences; the message should be adapted and delivered to each recipient according to his or her wishes.

- Just because Ellie is trying to deliver a message to Max, it doesn't necessarily follow that Ellie needs to know precisely where Max is, or even that he's not at his desk. In fact, Max may want to keep his whereabouts private except for when he specifically allows people to know what he's doing.

In sum, we need systems that can:

- **Respect people's right to control their own resources**, by ensuring that resources are controlled only by the resource's owner(s), and can enable restrictions on the resources as they see fit.
- **Respect people's privacy**, by limiting access to private information (such as a person's location), and only allowing such information to be retrieved from the outside by querying data sources controlled by the person in question.
- **Adapt to changing conditions**, so that the environment can adjust its actions dependent on what situation a user finds himself in, and
- **Be sensitive to individual preferences**, such that different people can guide the system in different ways based on their own personal needs.

How can we create the tools and infrastructure that can provide this level of robustness and adaptivity?

For one, we need a method of defining user preferences and using them to allocate resources such as displays or handheld devices based on the user's desires and the necessities of the current application. Ideally, such necessities should be able to make resource decisions based on qualities like *urgency* or *speed*. We also need to have an understanding of people's locations and their current situations, so that we don't end up intruding on a meeting at the wrong time. Such understanding also extends to maintaining knowledge of people's relations to locations and to each other, to determine, say, the grad students who are associated with the lounge. Underlying this is the need for storage structures that can organize all this information and easily retrieve the information we need.

We argue that an infrastructure can achieve the necessary levels of adaptivity and robustness by providing:

1. Agent communications and structures that mirror the social and physical interactions they represent.

2. A collection of user preferences, mapped into utility functions, being used to fuel the environment's decisions about service mapping and resource management.

3. An "awareness" of the state of the world, which can be used to further drive context-based preference decisions.

4. A semantic memory that defines and describes the resource and preference requirements.

The next section goes into more background on intelligent environments and explores these items in more depth.

Chapter 2

Introduction

As noted in the previous section, the entire field of ubicomp continues to grow rapidly. More and more research and development goes into technologies that embed processors in automobiles, medical equipment – even the components of furniture [4]. Indeed, Ray Kurzweil recently noted that Moore's Law is only the latest paradigm to show an accelerating trend, and that computing power per dollar has been continually expanding at a greater and greater rate, suggesting that the 21st century will contain a thousand times greater technological change than its predecessor[21]. Unfortunately, human productivity has not been changing at as quick a rate, and this suggests that we will soon reach a time when the critical resource in the ubicomp system is not the capabilities of the systems, but the people who use them.

2.1 Contribution

The fundamental contribution of this thesis is architectural, in particular it provides an architectural framework for human-centered pervasive computing environments on the scale of a large research lab or medium-size corporation. We regard the role of an architectural framework as three-fold – 1) It should provide modularity such that independent components with little knowledge of one another's details can nevertheless be assembled into a system exhibiting coherent global behavior, while 2) providing a reasonable level of performance for the system, and 3) guaranteeing that certain invariant properties, including the four mentioned at the end of the previous chapter, are respected throughout the system while requiring minimal effort from the developers of the individual components. What the architecture provides to enable this is a set of interface guidelines and infrastructure, a set of built-in components that manage the task of maintaining the global invariants.

In most cases, one tends to assume that the optimality criteria and invariants that an architecture will attend to are internal to the computational system, for example integrity of its data structures, or throughput of its processes. However, our concern is with human-centered, pervasive computing in which people play a central role. Thus, the issues of concern to us are those that deal with human social organization and the design criteria reflect the effectiveness and satisfaction of individual participants in the system. Thus we focus on two items – the ability of the system to adapt to individual users' preferences and the enforcement of a particular social norm: decisions over the use of resources should be made by those who control the resources, and decisions over how a task is performed should be made by the person involved in the task.

The system presented here, *Hyperglue*, provides infrastructure and interfacing guidelines that are able to achieve these goals. Of far less concern to us are classic questions of efficiency, since we see the critical resource in all future systems as being human attention while computational resources continue to be increasingly inexpensive. However, we believe that the structure we adopt also has classical computational benefits since it provides a modular structure that reduces the need to transmit information into global repositories or to engage in decision making with centralized single points of failure.

It has been noted to the author that in some ways, Hyperglue (or something like it) may be the "world wide web" for the rest of the processors in the world. Although the web provides a widely-used architecture linking the information in various computers together, it only works on a small percentage of the processors built each year; the rest go into automobiles, medical equipment, projectors, and other devices. As these processors take over more and more functions, and start being connected into networks of wider and wider scope, there need to be means to control and coordinate them effectively. We require an infrastructure that allows us to locate these devices, segment them into controllable chunks, and operate them. Hyperglue is a design that aims to achieve this.

2.2 An Infrastructure Overview

Recall the four pillars of infrastructure mentioned in Chapter 1.1:

1. Agent communications and structures that mirror the social and physical interactions they represent.

2. A collection of user preferences, mapped into utility functions, being used to fuel the environment's decisions about service mapping and resource management.

3. An "awareness" of the state of the world, which can be used to further drive context-based preference decisions.

4. A semantic memory that defines and describes the resource and preference requirements.

These represent the broad strokes of a design for a system that will, as noted, respect privacy for users and their rights to control their own resources, as well as adapt to changing conditions and be sensitive to variations in preferences.

To flesh out the design more fully, we divide agents into broad groups, based on the real-world entity for which they are acting. Such agent groupings are often termed *societies*, harking back to Marvin Minsky's definition of society in *The Society Of Mind* [30]. The real-world entity for which a society might act could be a user or a spatial environment. Where appropriate, we extend the concept to more abstract groupings of real-world objects, such as user groups or the environments that exist on the same floor of a building.

Once these societies are in place, we can use them to provide abstraction barriers that limit the need for communications that pass across the societal boundaries. When two different societies need to communicate, they do so by taking into account the preferences of the real-world entity they represent, and use that information to negotiate compromises that serve both sides well.

One means of requesting resources is to specifically request resources based on precise device descriptions. In order to handle negotiation correctly, however, we would like to move the mechanism of resource lookup towards one of *abstract service mapping,* which will allow agents to request, for example, a generic "display" service with contextual parameters. Doing so allows for a more flexible structure, and allows a cooperating society to provide many different possibilities to choose from when a service is requested.

Although the last item, the semantic memory, may seem to be an obvious component to those in the artificial intelligence community, it is actually one that is rarely explored within the field of intelligent environments. Most research into IE infrastructure relies merely on developing a simple repository of data, rather than trying to capture the meaning and relationships in a more general way.

In successive chapters, we will examine each of these components in more detail, and explore the design and implementation of *Hyperglue,* an agent-based system that uses these concepts to drive the development of a working infrastructure for multiple spaces.

We start, however, by examining a simple set of scenarios, and look at how existing IE infrastructure projects might handle these user needs.

2.3 Scenarios

As an example of the breadth of interaction that an IE should be able to support, consider the following scenarios:

2.3.1 Scenario 1: Talk Announcement

Alan is planning a practice talk for an upcoming conference. He fills out a simple form, and uses it to invite a few people (Beth, Charlie, and Diane) to attend the session and give feedback. Submitting the form causes a notification to be sent out to all three recipients, requesting the favor of a reply.

Beth is in a conference room. She has registered a preference to receive text messages as spoken utterances over speakers wherever she is, but as it happens, she's currently engaged in a small meeting in this room. Since her location is marked as being in "meeting mode", the system uses a different preference, popping up a small icon onto her laptop screen, which she can open at her leisure to retrieve the message.

Charlie is currently out of the office. His preferences state that he prefers email whenever he receives a message while out and about, so the system takes Alan's message and funnels it to his inbox.

Diane is out of the office as well. Normally her preferences are the same as Charlie's; however, she has added an additional preference that says that she should be contacted by phone if the matter is even mildly urgent. Since Alan's message indicates he would like a response as soon as possible, the request is routed to her phone through a text messaging service.

2.3.2 Scenario 2: Moving the Practice Talk

Alan is holding the practice talk at his desk, since only a few of the people he invited have actually shown up, and there's room for them to crowd around. The monitor on his desk is being used to display slides, as well as a small application which displays some useful utilities, such as a timer that lets him know how long he's spent on each slide.

After he's been talking for a while, some stragglers show up late. Alan decides to move the talk into a nearby conference room, where there will be more room for the participants. When he does so, the application he is running automatically reconfigures for the larger space, switching the slides to a full-screen, and allocating the slide timer view to a separate, smaller LCD screen. In addition, the application starts to take advantage of the greater capabilities of the conference room, using permanent cameras to capture a record of the talk.

2.3.3 Scenario 3: Free Donuts

This is essentially the scenario we describe in Chapter 1.1. Ellie is delivering a message to a large set of people, the exact membership of which is dependent on people's proximity to her current location, as well as their association with the location. For example, graduate students are chosen based on their membership in labs that are adjacent to a lounge, but also weeded out based on their proximity to the point in question.

2.4 Scenario Analyses

In existing systems for IEs, such as iROS[36], Metaglue[8], or OAA[25], performing these tasks is a matter of looking up an appropriate resource in some form of global directory, then sending a request to that resource. For example, an agent might try to locate a "message receiver" agent for each recipient, and upon finding one in the directory, would perform a "deliver message" request to complete the operation for each destination.

However, in order for the scenario to work in current IE infrastructures, a number of questions need to be considered. Let's examine some of the issues involved in these interactions in more detail.

1. *When Alan contacts Beth, how do we find out what devices are available in Beth's vicinity?* Several methods have been proposed for answering this question,

which often fall broadly into two categories. In systems that operate on a single-request model, such as MIT's Intentional Naming System [2], all devices that are in some way associated with Beth advertise that association to a global network, so that a single request can retrieve all of the appropriate recipients. Alternately, the methods can use two or more requests to get the information – Beth's software constantly advertises her location to the network, and Alan's agents must first look up her location, and then perform a separate search to find a set of candidate devices. Both of these methods require that **all devices be advertised globally**, and that **Beth's location be constantly advertised and updated** to reflect Beth's current situation.

2. *How can Alan and Beth share control of the interaction?* Once the agents working for Alan have located a device for the communication, we still need to make sure that Beth, or agents working on her behalf, can maintain some control over the interaction. Just as we accept an incoming phone call by picking up the receiver, Beth must have the ability to decide whether or not to accept an intrusion into her work. This can be done manually by Beth or, as in the scenario, automatically denying the use of a device under certain conditions. This ability should also extend to the point of being able to automatically refuse all interruptions for a period of time. In a device-centric model of the world, Beth would need to **continuously broadcast her preferences to any device she might use**, so that the devices would know whether to allow an incoming connection.

3. *How do we avoid conflicts with Beth's work or other activity?* Note in the first scenario that Alan's message appears on Beth's laptop. One can easily see a scenario where Beth's laptop is currently engaged (for example, projecting a slide during the meeting), and it would be inappropriate to take over even a portion of the display for a message. But how can Alan's agents discover which display is available for use? If the agents are using a global directory service to find the available display devices, then Beth's display devices must

be **constantly updating their availability status** so that Alan's agents can choose the correct resource. And what if Beth's handheld is unavailable? How can Alan's agents decide where to display the message?

4. *Who decides how information is presented?* As we've seen in the other parts of the notification scenario, recipients may have strong preferences about how they process information; for example, a short text sentence might best be conveyed as spoken utterances rather than through a display. If Beth were in her car, she may prefer that all information be emailed so that she can peruse it once she's arrived safely. Alan's agents would need to be able to discover these preferences, and alter their behavior accordingly. Doing so would require that Beth **make her preferences publicly available** to any agent system making communication requests.

5. *How does the agent system take into account the environment's access control issues?* Regardless of what the agents for Alan and Beth wish to do, Beth's conference room will need to have mechanisms in place to prevent unauthorized access to the devices by malicious agents. Without such a mechanism in place, one can imagine the IE's equivalent of "spam," where marketers are able to pop up advertisements on any available display space. The agent system needs to be able to filter out any unwanted requests, while at the same time giving Beth the necessary privileges to do her work and accept connections and data from Alan.

   Privilege is a double-edged sword; the data Alan is sending might be confidential or otherwise protected, and thus should not be displayed on the walls of a public, shared workspace like Beth's. **The environment itself has certain rules, and Beth needs to make sure those rules are respected.** In this case, Alan's agents must be able to mark the information as being removed from public consumption, and then have that request respected by the devices on the far side.

All the issues above point to a common problem; they are symptomatic of the lack of a modular structure that partitions the system's knowledge into well-suited chunks and allows the various software elements to deal with the knowledge appropriately. While modularity is often a hard concept to define, it at least is characterized by the existence of boundaries within which knowledge is encapsulated and which do not often need to be violated. These boundaries also provide a sense of coherence to the knowledge within the boundaries. Thus, our biggest challenge is to find an appropriate modular structure for the system's knowledge. As we develop such a structure, we are also guided by concerns for four challenges: scalability, access control, customizability, and adaptivity:

2.4.1 Scalability

A common trait of IE infrastructure systems seems to be a central coordination directory service like Metaglue's CatalogAgent or OAA's Facilitator[25]. When dealing with the simple case of a single user operating in a single environment, there doesn't tend to be large numbers of similar agents, and so this design can perform reasonably well.

However, such a design will inevitably yield scalability problems when the full needs of highly distributed environments enter the picture. For example, many of the agents that were working for the single user or the single environment now need to be duplicated for each user or for each space that might want to communicate. Similarly, there are cases where an environment may have many different methods of fulfilling a request, and for each of these methods an extra agent may be inserted into the directory. A typical environment can have well over a hundred agents operating within it, and a building might be populated with a hundred or more such environments. If we want to have different environments communicate between buildings, we might soon enter a realm where there are hundreds of thousands of agents trying to use a single directory service. As a result, network contention can cause access times to rise, and the larger directory sizes can cause lookup times for each access to rise as well. Although clever tricks in engineering might alleviate some of these concerns, the central service acts as a choke-point on the system.

Similar problems occur with a publish-subscribe model such as iROS's Event-Heap architecture[36]. As the number of environments and users increases, so does the number of publishers, and the heap must do extra work to handle the larger number of requests and parcel information out to the larger number of subscribers.

As we move towards larger multi-user systems, we need a better solution to this problem. We propose that one way of handling this more efficiently and avoiding the scalability issues is to group agents together so that agents are directly aware only of the small subset of agents that they work with most closely, and other agent communication is funneled through access points for each group. I refer to this as the "society" model, and describe it further in Chapter 4.

2.4.2 Access Control

As noted in the questions above, access control issues abound, since there is often a dichotomy between the desires of the person who wants to send information and the desires of the recipient. For example, the sender may wish to utilize a speedy connection to ensure that the recipient sees the information immediately, whereas the recipient may not wish to be constantly distracted with information that is irrelevant to his or her task at hand.

To address these concerns, the agent system must provide methods of access control and negotiation, which will allow agents to guide communication and interaction so that one user cannot impose behaviors on another's agents without proper consent. A large part of this is also handled by compartmentalizing agents through the society model (described further in Chapter 4). Additional pieces are provided through the creation of abstract services (see Chapter 5) that can, if needed, select specialized resources based on access limitations, or deny access outright.

2.4.3 Personalization and Customizability

The scenarios often make use of the user's preferences. People tend to have different approaches to their work and communication methods; the IE should respect those differences, and allow the user to guide the system by specifying device preferences or otherwise customizing their working environment.

In addition, customizability of applications allows us to perform application reconfiguration on the fly, so that displays can either be shared or reorganized to best make use of the devices that are present.

The abstract services (Chapter 5) briefly mentioned above can perform part of this task, but will need to coordinate with a storage mechanism that records user preferences and acts on them (see Chapter 6).

2.4.4 Adaptivity

The system must be able to recognize changes in the state of an agent-controlled object, and adjust to those conditions. Imagine, for example, a projector whose bulb has exceeded its lifespan, and hence gone dead. If an agent in the system desires this resource, there must be a means of either shifting control to a less desirable resource, or of failing gracefully and informing the appropriate entity of the condition.

For this, we need to be able to have resources that react to the current contextual parameters of the world. In addition, we require both the aforementioned abstract service definitions (Chapter 5) and a semantic memory that can record the contextual parameters for the selections (Chapter 6).

2.5 Infrastructure Overview

To meet these four challenges, I have designed and built an agent-based software system that is capable of handling the interactions in the described scenarios. The individual components of this system will be described more fully in the following

	Scalability	Access Control	Customization	Adaptivity
The "society model"	X	X		
Preference mapping		X	X	
Context Handling				X
Semantic Memory			X	X

Figure 2-1: A small table mapping issues in intelligent environments to the components of this infrastructure that most fittingly addresses each of them.

chapters, but let us briefly outline the components of the architecture:

- Agents organized into *societies* that have counterparts in the physical world, whether that counterpart be an environment or a person (or aggregates of these).

- Each of these societies contains agents that perform several vital functions. These include: resource management, which can take a simple call for a piece of functionality and translate it into an agent that can fulfill that function; service mapping, which can take higher-level requests and turn them into collections of agents or plans to perform more complex tasks; semantic data management, for storing and querying information about the world; and local catalogs for managing the agents in the society.

- In addition, each society maintains an *ambassador* which can locate and communicate with the other societies, turning local requests into remote ones by forwarding them when appropriate.

Let's briefly examine how this architecture can be used in the messaging scenario. First, there are agents collected into societies for Alan, Beth, Charlie, and Diane. Alan's society (or more precisely, the agents within it) first formulates a request for a "message delivery" service. This service has certain parameters attached, specifying that the message is textual in nature, and indicating that the message has a low level of urgency. The same request is prepared to be sent to Beth, Charlie, and Diane.

Alan's society recognizes that Beth, Charlie, and Diane are all separate societies, so his service mapping system delegates the request to the remote societies. As such, his ambassador locates the ambassadors for these separate societies, and then forwards the same "message delivery" request on to each of them.

Charlie's ambassador, for example, receives the request, and then forwards it on to his local service-mapping engine. Charlie's engine uses the local semantic data storage to examine Charlie's personal preferences, and uses them to find a delivery plan which satisfies the parameters. A set of suitable plans are returned to Alan's service mapper, which chooses an appropriate plan incorporating Alan's preferences.

In this case, Charlie's returned plan is just a simple "deliver a message" plan, which requires a simple delivery resource. Alan's service mapper executes the request for this resource, which is again forwarded to the remote society. Charlie's resource manager receives the request, uses the semantic data store to recognize that Charlie is currently out of his office, and thus provides the location of an agent that delivers the message to his e-mail. The ambassador provides a handle for this agent back to Alan's society, and Alan's agents then deliver the text directly through this handle. Charlie's email delivery agent then takes responsibility for providing the information to his inbox.

In effect, all the issues that we need to deal with are concerned with the management of the system's knowledge including its modularization, distribution, communication, encapsulation and storage. Having the right knowledge in the right place facilitates adaptivity and context sensitivity. In looking at these issues we need to be concerned with scalability (how do we manage large amounts of changing knowledge without an unreasonable amount being in any one place) as well as access control (how to avoid divulging information that one would prefer be kept private and that doesn't need to be divulged).

We deal with these issue by structuring the system around agent societies each equipped with facilities for remote communication, semantic data storage for recognizing and acting on contextual elements, and service mapping and resource management engines that use the context appropriately; and that work with one another by finding a service mapping that best satisfies their mutual preferences.

The following chapters explore existing agent systems, and then move on to describe the thesis components – society model, service mapping, and semantic memory – in more detail, discussing details of an implementation that allows the functionality in the scenarios to be accomplished.

2.6 Roadmap

As a "roadmap" to the rest of this document, let's briefly examine Scenario 3 in detail and provide pointers to the chapters or sections of the thesis which define the necessary infrastructure in more detail.

Ellie starts by instructing the lounge's kiosk to tell nearby people that food is available. The kiosk's user interface translates this spoken utterance into a request to send a "food is available" message with appropriate parameters which describe the necessary urgency (not very) and timeliness (high, because the food might soon be gone). This request is handled by whatever software is running on behalf of the lounge (Note that when this section refers to the lounge or another non-mobile entity performing an action, this is what is intended.)

The software associated with the lounge contains knowledge describing its relationship to the other groups in the area, based on a semantic network (detailed in Chapter 6). Included in this knowledge are indications that there are a set of groups associated with this lounge, so the knowledge base contains assertions such as "the AIRE group is adjacent to Lounge 32-241" and "TiG is adjacent to Lounge 32-241."[1] Using this information, the lounge identifies a set of groups that should receive this message.

At this point, the local lounge must forward the request on to the different

[1]A more thorough system might actually just separate the physical layout of the building from the group assignments, saying that "room 32-221 is adjacent to room 32-241" and "Group AIRE is in room 32-221", and rely on an underlying system to bind those two assertions into the group-adjacency relationship. For this section, we assume that such a group-adjacency relationship can be easily found.

groups. First, it must locate software running on behalf of these groups in order to communicate with it, and it does this by using a directory service that can find a "society" acting for those groups (detailed in Chapter 4).

From each of these groups, the lounge requests a message-sending service, providing the necessary service parameters that were needed in terms of urgency, etc. Using these parameters as a guide, the lounge and group societies determine the most appropriate service (detailed in Chapter 5) and then the lounge executes the agreed-upon service by providing the message as an argument. The actions for performing this can be accomplished through any agent framework (an example of which is described in Chapter 3).

In the case of the individual groups, there is really only one service that makes sense (no matter what the service parameters) – one which multi-casts the message to the individual members of the group, keeping the same service quality parameters that were originally requested. To this end, the group then performs the same operation that the lounge did – locating a society for each user, and working with the user's service request engine to determine the best possible means for sending the message on to the user, and then executing the service with the appropriate parameters.

As such, the AIRE group's society first locates Max's society and requests a message-sending service with the above parameters (low urgency, high timeliness). Max's society knows Max's current location – even if that knowledge is limited to simply being aware that he's not at his desk. This society also has a number of means of getting a message to Max; the message can be sent to his phone, sent over email, or displayed on his desktop. Since Max is not currently at his desk, the quality of service for phone service is high timeliness mixed with high urgency; email has a medium timeliness, and desktop displays can only guarantee low timeliness and low urgency.

Since Max has set up preferences (see Section 5.2) that state that he receiving messages quickly is more important than concerns over the urgency or the costs of using the cellphone, the service mapping engine selects and provides a pointer to Max's phone delivery service as the most appropriate service, and provides a handle to the service to the requesting AIRE society. AIRE then uses that service to forward the message on to Max.

Chapter 3

Metaglue – An Overview

Before we begin to explore more fully the details of the design outlined in section 2.5, let us first examine existing IE agent systems, and in particular the Metaglue framework developed at MIT. The Metaglue framework will be explored in more detail because it serves as the basis for this thesis's work, and was used to show how the Hyperglue design can be implemented by modifying a proven agent system. Thus, this thesis can also serve as a model for extending other agent infrastructures, such as Open Agent Architecture (OAA), iROS, or Hive, along the same ideals.

3.1 Notes on Agent Systems

There are a variety of different distributed agent systems being developed and used for intelligent environments. Among these systems are OAA [25], the Hive agent system [29], iROS and the EventHeap architecture [36], and the Java-based Metaglue [8] system developed at MIT. All of them – indeed, distributed object systems in general – provide functionality for performing several basic tasks:

Name Mapping In order to reference objects that could exist anywhere in the network, a naming convention and lookup mechanism is required. Because of this, agent systems require the presence of an ontology (whose level of formality – or lack of formality – is often defined by the system's design), which can be used to provide a simple means for agents to refer to objects. The lookup mechanism provides a means of taking such ontological references and translating them to network addresses or other means of accessing the object across the network.

Object Creation Since such systems normally rely on highly distributed operation, facilities for instantiating objects on the fly are a necessity. This can either be done with explicit calls by the objects in the system, or be performed implicitly the first time an object is referenced.

Remote Communication In order to actually communicate with a remote object, facilities must provide some transport for passing data requests and results across the network. Essentially, this boils down to some form of remote procedure call mechanism, although many object systems provide for a more object-based approach such as the Common Object Resource Broker Architecture (CORBA) or Sun's Remote Method Invocation (Remote Method Invocation (RMI)). Such an object system must usually provide means for automatically "boxing" and "unboxing" the parameters of the remote method calls so that objects can be successfully copied from one network location to another.

Higher-level systems may also provide means for:

Semantic Discovery Rather than doing simple name mapping, it may be possible to search for and locate objects using a more flexible semantic description of the object. Sun's Jini[27], for example, accomplishes this using a set of strongly typed attribute values; each of which provide a many-to-many mapping between objects and a set of attribute values [28]. This allows a single object in the system to have multiple names in different languages, for example. Discovery operations within Jini are performed by providing a template of attributes, and returning the object or objects which will satisfy the template's constraints.

Method Selection The flip side of semantic discovery, which allows for richer means of selecting objects, is a richer method selection mechanism, in which semantic descriptions of the desired process allows for a higher-level means of describing methods and choosing the most appropriate one.

These higher-order means of communicating with agents are useful, but are not strictly necessary for the programmatic control of an agent system. However, as the capabilities of an intelligent environment (IE) grows, being able to provide this kind of semantically rich discovery and selection operations to the user is necessary, so that the user is somewhat protected from having to recall the names and operation of all the applications in the system. Whether these semantically inspired tasks are built into the system, or layered on top of a lower-level agent framework, their capabilities need to be provided to the user at some point.

3.2 Metaglue Design and Operation

The basis for the AIRE Project's [3] agent implementation is Metaglue [8], a Java-based agent system that provides for the easy creation of agent networks, distributed amongst many different platforms.[1] Here I describe the Metaglue system in some detail, and describe some of the enhancements made to the system since the initial design by Phillips [34].

Metaglue makes extensive use of Java's design and RMI components to provide the basic substrate for the agent systems. Name mapping is performed utilizing the RMI registry, and the basic ontology grows out of the standard means for identifying Java classes. Similarly, the communication transport utilizes the RMI design. Metaglue does not have higher-level frameworks for semantic-based discovery or method selection (but we explore how to add those in in Chapter 5).

Like all Java objects, each agent running under Metaglue runs on a Java virtual machine (VM). In addition, any VM running Metaglue agents must first start

[1]Much of the basic information on Metaglue is found in Coen et al.[8], although the focus there is describing Metaglue as a programming language and toolset rather than as an agent system.

a singleton instance of a special platform agent, called the MetaglueAgent[2]. The MetaglueAgent is responsible for communicating with the CatalogAgent, a separate agent which, like the RMI registry, holds information about all agents in the running system. The MetaglueAgent is also responsible for starting and stopping other agents on the local VM on request.

3.2.1 Agent Identification

Every agent needs to be named with a unique identifier, so that it can be queried and located by the agents running on the Java VM. These identifiers are stored in the CatalogAgent, and retrieved on request.

The naming scheme used by agents in Metaglue consist of AgentID objects, each of which contains three fields: a society, an interface name (which in Metaglue is called the "occupation") and an optional unique identifier (called the "designation").

The society is a simple string, which Metaglue uses as a naming and scoping mechanism for agent divisions. The term "society" is here taken from Minsky's Society of Mind theory [30], and is intended to collect agents together which are operating for a common purpose. One way that this field can be used in the AgentID is to define the entity for which the agent operates. In practice, this is usually something like a username or set of initials for a user, or a room name and number (such as office832) for an intelligent space.

The agent occupation is a name for the API that the agent implements. For simplicity's sake, this usually references the Java interface name that the agent handles – for example, a projector might have an interface name like device.display.Projector.[3]

Finally, the designation is used to disambiguate individual agents that share a

[2]Early papers on Metaglue (including Coen et al. [8]) refer to this agent as a "Metaglue virtual machine," or "MVM" for short. Since the term "virtual machine" often implies a larger architecture executing a defined instruction set, I prefer to use the term "platform" in this thesis.

[3]All of the core agents in Metaglue exist under the "edu.mit.aire" namespace. For purposes of readibility, this thesis will often omit that common portion of the naming convention.

society and an occupation, and thus could be confused. For example, a conference room might have multiple projector displays available for use. Each one must have a separate designation, either a name or perhaps a number, so that they can be addressed separately.

AgentIDs are often referenced by strings, with the individual components separated with simple punctuation characters. One projector might be fully identified with office832:device.display.Projector-north, where "office832" represents the society, and "north" the designation.

3.2.2 Agent Communication

In order for one agent to communicate with another, it must first obtain a "handle" that will mediate the communication with the destination agent. This is done by contacting the Metaglue catalog and requesting the handle for the destination's AgentID.

Metaglue is based on Java RMI, so these mechanisms make extensive use of the underlying RMI facilities; in particular, the Metaglue catalog is closely tied to the RMI registry. In the Java RMI framework, a running object registers itself with the registry under an arbitrary key. Other objects then consult the registry and obtain a handle by presenting that key to the registry. The registry then returns the handle (called, in RMI parlance, the "stub"), to the requestor. The stub is a lightweight object representative which relays method calls to the stub across the network to the actual running object, and return any results back to the local caller. This is a minimal mechanism intended to facilitate the construction of a variety of richer interfaces, while making remote object communication as straightforward as calling methods locally.

Metaglue provides several capabilities on top of RMI; the two most important of which are automatic starting of agents and guaranteed reliable communication. Both of these are provided through the Metaglue reliesOn primitive. When reliesOn is invoked using an AgentID, the Metaglue system checks whether the specified agent already exists, and, if so, returns a handle to it. Otherwise, a new object satisfying the API specified in the AgentID's occupation is created, a handle for it is stored into the Metaglue catalog, and the handle returned to the caller.

The second goal of guaranteed reliable communication is provided through *proxy* objects (called error-handling avatars, or EHAs) which are created locally during the reliesOn call. All communication to the remote agent is funneled through the EHA proxy object. The purpose of the EHA is to easily handle communication errors and failover conditions without having to force each agent to perform special processing on its own to detect agents that are unavailable due to a crashed computer or frozen process (originally specified in Warshawsky [43]). The EHA proxies automatically retry communications in the event of network errors, and even restart agents that become unavailable. All this functionality is invisible to the agent programmer, and ensures that the agent will complete any request to the destination.

The proxy also makes it easier to lazily start up agents – the reliesOn call returns immediately, even if the remote agent is not present and needs to be started. While the remote agent is initializing, the calling agent can continue processing, and the EHA will only block and wait for the remote agent when it requires a remote method call to complete. This is especially useful since Metaglue agents may take some time to initialize; if an agent needs to use a separate agent for handling a user interface, it behooves us to try to start them in parallel as much as possible. In these cases, the agent's initialization routine would first call reliesOn to start up the user interface agent, and then continue with its own initialization. When the time comes to call a method on the user interface agent and populate the UI with specific data, the proxy automatically pauses the method call until the user interface agent has completed its own initialization routine successfully.

Because the proxy provides a level of indirection between the local agent and its requested communication partners, any requested agent can be replaced at runtime with another that satisfies the same interface, simply by changing the destination that the proxy communicates with. This allows for dynamic replacement of agents in a running system, without forcing agents to re-send their reliesOn call. This is used often by Metaglue's resource management architecture to substitute new agents for old ones without an expensive restart of all the agents in the network, and is a key component of the Hyperglue system.

3.2.3 Notification Systems

Although the one-to-one communication provided by the reliesOn primitive is useful, there are many times where an agent needs to employ multicast messaging, to deliver a single message to a multitude of destinations. For this, Metaglue provides a publish-subscribe interface called the NotifierAgent. Agents send notification updates with an identifier to the Notifier, which acts as a clearinghouse for the updates and passes them on to all agents who have registered for updates concerning that identifier. Similar mechanisms can be found in other agent systems, such as OAA or the EventHeap architecture of iROS.

In Metaglue, the Notifier works fairly simply. All agents that wish to broadcast information do so by providing the data along with a string-based identifier that describes the type of information being sent (usually the application or device name, followed by an information category, such as "map.select"). By convention, different components of the identifier are separated by period characters. [4]

The agents that wish to receive notifications send a request to the Notifier, providing an identifier of the kind of information the agent wishes to receive, and the name of a callback method that should be used by the Notifier. The identifier in the request may also contain an asterisk wildcard ("map.*") to indicate that they want any information from a given application. If an agent wants to receive all notifications (a rare occurrence except for debugging applications), it does so by simply requesting just the wildcard "*" identifier.

When an agent publishes its message information to the Notifier, the identifier

[4]In Metaglue parlance, the information and identifier are collectively referred to as a "secret." Because this name is rather inappropriate for an item that gets passed to anyone who wishes to hear it, I will use the term "notification" to describe the message information.

is checked and the notification is then forwarded on to all agents that had previously made matching requests.

Many Metaglue agents use the Notifier extensively, for a variety of purposes:

- *To monitor changes to the world state.* This is a common usage. For example, a map-display agent in the room publishes notifications whenever the current display changes. This is used by an associated speech-processing agent to determine what locations are currently visible on the map display, and uses this information to prepare for spoken queries involving the visible locations. Device controllers for lights publish notifications whenever the lights turn on or off, so that other agents can monitor an environment and make assumptions about the current lighting levels. The Metaglue catalog also publishes notifications describing the current state of the agents for debugging applications.

- *To manage server-client architectures.* Some agents operate on a client-server architecture, where many different agents communicate with a central agent to coordinate and share information. One example is the "intelliCD" music application, which allows several clients to monitor and control the ambient music playing in a shared environment. Although each client sends its requests directly to the server – requesting a new song, or expressing a like/dislike reaction to the current music – the music server in turn uses the Notifier to broadcast the collected reaction information to the clients, as well as sending updates about the song that is currently playing.

- *To create flexible user interfaces.* In order to create mobile interfaces that are capable of being shifted to new locations on the fly or even replicated on many different displays, Metaglue agents are often designed to separate the control logic from the graphical user interface. When this is done, the interface is usually embedded into a Java object and sent to a GuiManagerAgent which uses notifications to receive information from the control logic and update the user interface.

3.2.4 GuiManagers

The GuiManagerAgent design is worthy of more explanation since, although it falls short of enforcing a true Model-View-Controller design, it does a good job of separating the main program logic from the interface and easing the creation of portable, flexible user interfaces in an agent framework.

An agent that wishes to publish a user interface will package the interface into an object called a GuiMaker. This is simply a Java object that satisfies some simple conventions – it can manage its state so that it can be transferred over a network connection, and it has the ability to take a section of screen real estate and render the interface inside of it (called a Container in Java parlance). In addition, it shares some aspects of agents without being an agent itself – it can listen for notifications and send them out, and it can use reliesOn to procure a handle to an agent in the system. The GuiMaker does this by cooperating with a management agent (the aforementioned GuiManagerAgent) to deal with notifications and make requests from other running agents.

Often, the interface will not maintain any state variables of its own, relying instead on querying the agents directly for information. Thus, the interface objects are fairly simple code, directly translating an action on the interface into a method call on an agent running in the system, and using the agent to manage any important information.

To make this work, the agent framework starts instances of the GuiManagerAgent on all platforms that are tied to displays. Thus, when an agent wishes to show an interface in a given location, it contacts the manager for that display, and passes the GuiMaker for the interface to it. The GuiManagerAgent then takes responsibility for displaying the interface and mediating communication between the agent framework and the user interface; it will do this by passing notifications from the agents to the user interface, and allowing the user interface to run reliesOn and make requests of agents.

Imagine, for instance, a simple CD player control (see Figure 3-1). In this case,

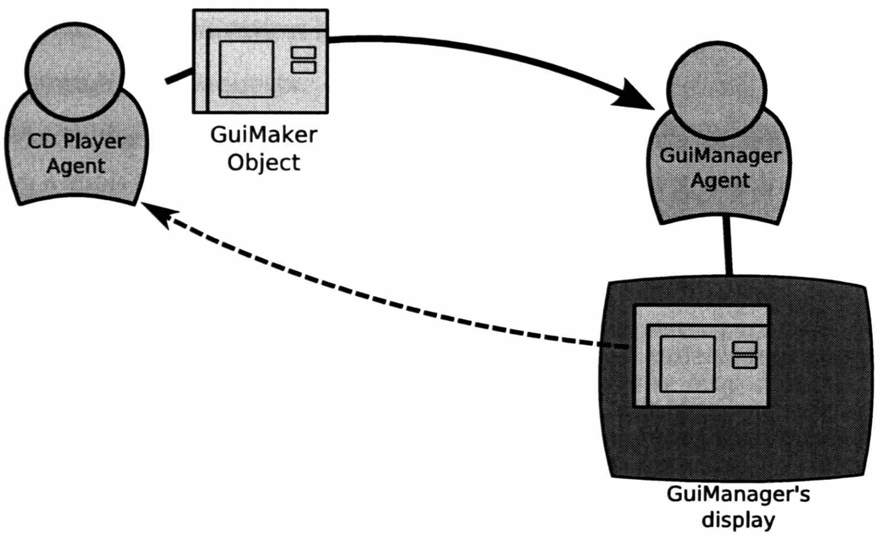


Figure 3-1: A CD player using a GuiManager for display. The CD agent passes an interface object to the manager's display, which can then be used to pass event requests back to the originating agent.

there is an agent that can directly control the CD player, and a GuiMaker object consisting simply of a text display that can report on the currently playing song, and a handful of buttons that allow the user to pause the song, skip to another one, or control the volume. As the agent starts, it delivers the GuiMaker to a manager running on an appropriate display, and the manager will launch the UI code in the GuiMaker. The GuiMaker object will then query the agent to determine the currently playing song, and perform requests when the buttons are pressed. In turn, the CD player agent sends notifications announcing updates to the currently played song, so that the interface display is kept up-to-date.

This separation – putting state information into the agent and user presentation into the interface – provides several possible scenarios for dealing with user interfaces to agents:

- Portable interfaces, which can be moved seamlessly from machine to machine. Because most interfaces have no internal state, this translates to the action of closing a window on one machine and opening it on another.

- Replicated displays, which appear in many different places.

- Multiple presentation interfaces which can display data from the same agent or agents in different ways, based on either the capabilities of the output device or the user's desires.

- Shared displays, in which the GuiManagerAgent can tile or overlap GuiMaker objects from several agents into one large window.

Because of this separation, developing portable applications that work well under Hyperglue is far easier. This will be explored further in the next chapter.

3.2.5 Security Hooks

The proxy system described above is also used as the basis for a security system with method-level granularity – that is, security systems can create a version of the proxy which refuses to send a message under certain conditions. Any agent can specify its own version of the proxy to be used for communicating, and its proxy can refuse a call to any method for any reason. More details of this are found in design work by Kottahachchi [19].

Chapter 4

Extending Agent Systems Around the Society Model

The first pillar of this thesis consists of modeling agent communication structures around the social and physical interactions that they represent. In this chapter, we describe the design of the agent society model, and show how we use it to structure communication in keeping with that model.

As a brief reminder, by the term *agent* we mean a software object that exists in the context of a distributed system, capable of remote invocation by other agents, and acting on behalf of an entity in the real world. Agents can operate at a very low level in the system – for example, as a simple device controller – or be a high-level object that embodies a strategy or plan for marshalling lower-level agents to perform a task. What all agents share is that they operate within a common framework, which provides mechanisms for agent discovery and communication (either directly from agent to agent or indirectly through a publish-subscribe or blackboard architecture).

From some perspectives, this often means that a collection of agents have greater differences (their level of abstraction, the knowledge that they maintain or have access to) than similarities. When this is the case, we try to use qualifying terms ("the planning agent") to make the distinction; otherwise, when our concern is only with overall protocols or mechanisms that all agents adhere to, we use the term agent indiscriminately.

4.1 Communication the human way – Scenario I in real life

In scenarios like those described, there might be a large set of agents all operating at once. Some of these agents may actually be simple, low-level device handlers, such as a controller for a display or an e-mail processor. Others are more high-level abstractions, for example, Alan's agent that gathers the list of recipients and sends out notifications. A system running just the notification scenario in Section 2.3.1 can easily encompass a hundred agents, all working in concert to provide the needed functionality. Add in more tasks for the IE, and expand the number of users that the system needs to be capable of handling, and the number of agents skyrockets.

When dealing with large distributed agent systems, there are often scalability concerns. Performing agent discovery, for example, requires access to some sort of directory lookup mechanism, which needs to be able to find the appropriate agents without significant slowdown. In addition, failover conditions for any agent in the network need to be handled gracefully.

There have been several approaches to handling these problems (for example, the Intentional Naming System [2]), but these often have scalability issues lurking behind the scenes. For example, the initial revisions of INS assumed that all the individual name resolvers would know all services that they would ever need to route to, effectively causing all the resolvers to maintain a global knowledge base that was continually propagated throughout the network. Where a small organization might be able to maintain this information easily, as the number of services grows, propagating and maintaining this information became an expensive proposition.[24]. In addition, the design of the resolver's data structures become inefficient when the number of service entries climbs into the realm of tens

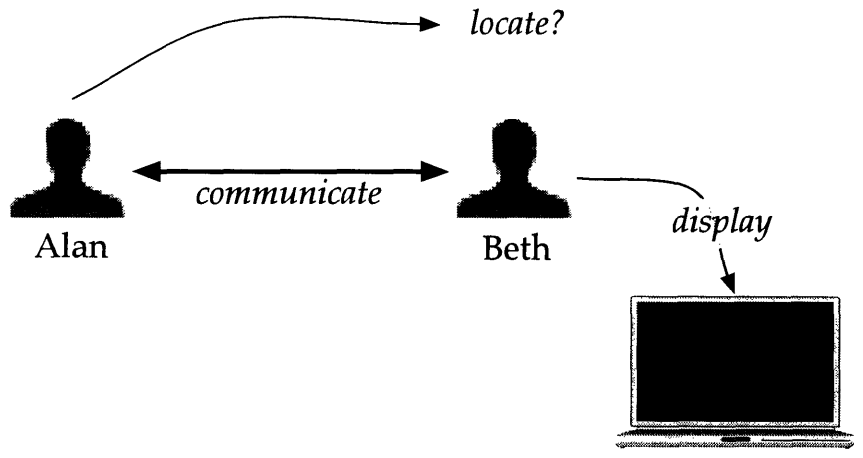


Figure 4-1: Sending messages from Alan to Beth – human version

of thousands, so INS becomes inadequate for maintaining service information for very large organizations.

In addition, there are security issues to be dealt with – if any agent can contact any other agent in the system, then an appropriate negotiation needs to take place to ensure that an agent operating on behalf of one person isn't being used inappropriately to retrieve information to which it shouldn't have access.

When examining the portion of the scenario that deals solely with Alan and Beth, we can divide the agents into three broad categories: those working for Alan, those working for Beth, and those working on behalf of the conference room. This provides us with a roadmap to simplifying the communication and scalability problems, by organizing communication along the same pathways that humans would naturally use.

Imagine that the simple message-sending scenario with Alan and Beth were being handled directly by the parties concerned, rather than through agents:

1. First, Alan would ascertain Beth's current location.
2. He contacts her through some communication medium.
3. He lets her know that he wants to send her some information, and asks the

best way to get information to her.[1]

4. She responds with her preference (e.g., "send it over IM"), and Alan either follows her wishes or negotiates a better method for sending the data.

5. If the channel is something that might require a larger display – for example, if Alan was sending a URL to a web page that Beth wanted to display on a wall of the conference room – Beth would have to work directly with the devices of the room, directly making decisions about which devices are free or in use by others in order to display the information appropriately.

To restate the above in broad strokes, Alan contacts Beth, negotiates for a message-sending interface, and uses it to send information. Beth contacts the devices in her current location to find a display interface, and uses it.

4.2 Societies of Agents – People and Places

Looking at the human-centric version of the communication model, it's easy to see the difference between it and a fully ad hoc approach where any agent in the system has global knowledge of other running agents. In essence, all interaction is funneled into communication between people, or interactions between a person and the devices at the person's location.

This yields a method for mitigating many of the scalability and security concerns in the agent communications. Essentially, we cluster all agents into a group based on the real-world person or place for which they act (Alan, Beth, the conference room, etc.). Following Marvin Minsky's terminology in "The Society of Mind"[30], we will hereafter refer to such a group of agents as a *society*. All of the agents are categorized as belonging to Alan's society, Beth's society, or the society for the conference room. In addition, we also designate one specific agent to

[1]If the information is just the notification that we see in the scenario, of course, Alan would probably short-circuit the process and just tell Beth directly what he wished to say for the sake of expediency. When we translate this example back to the agent-based world, however, these desires for expediency are better handled through agent negotiation.

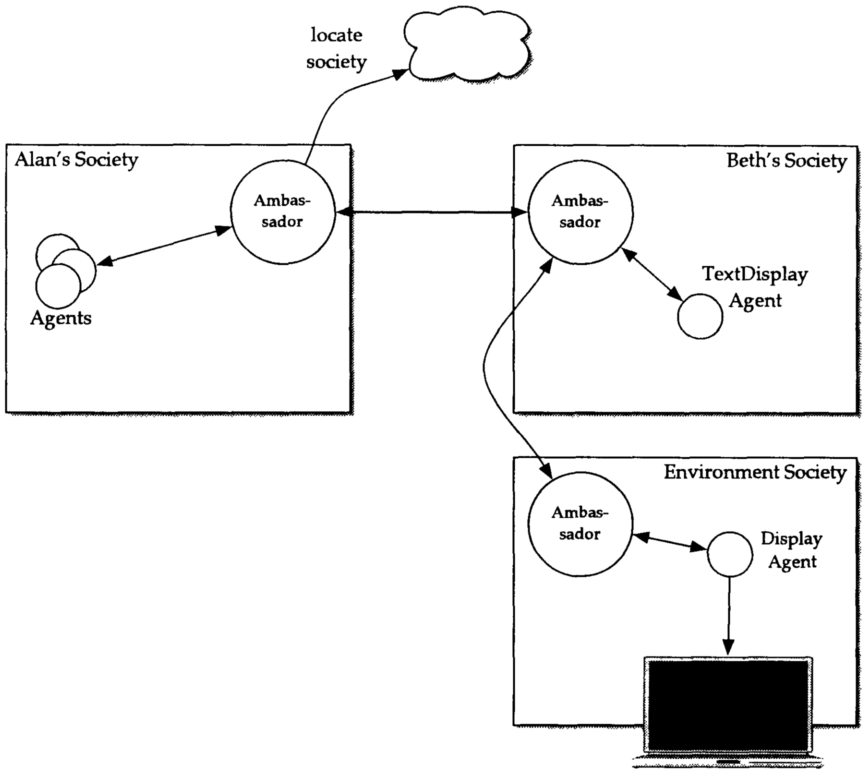


Figure 4-2: Sending messages from Alan to Beth – agent-space analogue

handle the communication flow, thus abstracting all a society's agents behind an "ambassador" agent, which in turn mediates communication.

At this point, the agent-space analogue to the human-centric communication model becomes a straightforward translation of the above:

1. Agents working for Alan use the local catalog to contact Alan's ambassador, which in turn uses a directory service to locate the ambassador for Beth.

2. Alan's ambassador opens a communication channel with Beth's ambassador.

3. The two ambassadors engage in a handshaking protocol, and Alan's ambassador requests a service for sending information.

4. Beth's ambassador returns some candidates for selection, or alternately might open a negotiation with Alan's ambassador to find the best possible candi-

date.

5. If the selected service requires communication with the local environment, Beth's agents perform a similar operation with the ambassador for the local environment's agents, requesting a service for graphical or auditory representation of the information.

What benefits does this approach yield?

- There is no need for a globally-published listing of all agents; instead, the problem is reduced to the need to find the societal ambassadors – a much smaller set. This reduces scalability concerns in any global directory service. In addition, it mitigates at least one security concern by preventing rogue agents from "surfing" the global listing to find vulnerable services, since the full listing of agents in any society is not provided to the world at large.

- A potential security checkpoint is now available through the ambassador. Because all external communication is mediated through the ambassador, the ambassador functions also as border control for the agents, denying access to agents unless they are coming from an approved source.

- Coding of individual agents is simplified. Because the ambassadors are tasked with many of the more complex duties of the distributed communication, the individual agents are relieved of many of these responsibilities, and instead can focus on their own individual operation. In addition, because all communication requests come from a local source (the ambassador), the individual agents treat all requests as if they were locally generated.

- It makes it easier to unite heterogeneous agent systems. Because the details of any agent network are now abstracted behind the ambassador agents and public APIs, allowing one agent system to talk to another simply requires that the ambassadors be able to communicate and pass API information back and forth.

There is a cost to this structure, of course. A simple global list of every agent in the world is a much simpler means of finding information, and the ambassadorial structure will incur costs in processing time, as requests get routed through the ambassadors. I submit, however, that the benefits of this approach outweigh the minor impediments to performance.

4.2.1 Ambassadorial Functions

With this design, there are several functions that need to be handled by the ambassador agents in order to provide the benefits mentioned.

Real-World Entity Representation

Because all agents are operating on behalf of some real-world entity, and the ambassador is the central representative of each agent society, the ambassador also acts as the representation in agent-space for the real-world entity. As such, it can deliver answers about the real world, such as location information, to agent-space objects.

Border Control

Although the individual agents in a society, or the society itself, can create their own means of dealing with access control, the ambassador acts as a first line of defense against malicious requests, simply by not allowing requests by agents that do not identify themselves as being trusted. This can be as simplistic as having the ambassador maintain a list of agent societies that are trustworthy, or can run to a more intricate access control mechanism utilizing role-based access controls.

Negotiation of Agent Services

Since the ambassador is the main conduit between external societies and the local society, it is well suited to negotiate with the external society regarding the service qualities. Whereas local services are usually chosen based on the preferences of the entity the agents are working for, finding a service that balances the needs of both the local and remote agents is more difficult. The ambassador is used to offer and negotiate for the best results.

Details of the local service mapping case and service qualities are described in Chapter 5.

4.3 Hyperglue: Applying Societies to an Existing Multi-Agent Framework

Let us now examine how we take the society-model design for agent communication and apply it to an existing multi-agent system. The first version of Hyperglue is implemented atop the MIT Metaglue multi-agent framework. Chapter 3 described the Metaglue system in more detail; it may be worth referring to it for more information.

4.4 Building Hyperglue

Although the Metaglue system provides a foundation for agent discovery, communication, and resource management, its shared catalog makes it less than ideal for our scenarios in Section 2.3. Specifically, it doesn't provide adequate separation between different users and spaces, and makes it far too straightforward for a malicious agent to take control of a space without regard to the space's policies and procedures.

To alleviate these, we abstract the discovery, knowledge representation, and resource management tasks to a higher level, treating each society of agents as a single entity, and looking at the interactions between societies rather than agents. We refer to this higher-level abstraction as *Hyperglue*, the first version of which is now deployed in the AIRE Project's laboratory.

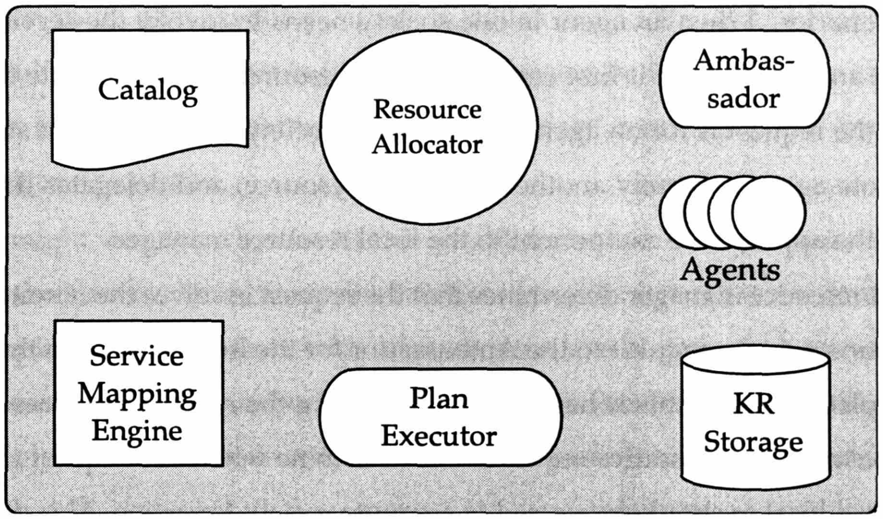


Figure 4-3: The elements of a Hyperglue society

4.4.1 Overview

Hyperglue provides a communication interface between agents, situated at the level of "real-world" entities such as people, places, and organizations. In this section, we illustrate how an agent that is a member of one agent community uses resource management systems and Hyperglue's entity discovery capability to communicate with an agent in a different agent community.

We specify each agent community by extending the Metaglue notion of a society. Whereas in Metaglue the notion of a society is little more than a grouping construct, Hyperglue introduces more required structure into each society (see Figure 4-3). This includes a catalog of the local agents in the society, service mapping and resource management subsystems, ambassador agents for communication external to the society, and knowledge engines that can drive local decision making. In short, each society acts as its own miniature agent system, only dealing with external societies when necessary.

Consider two agent societies that need to communicate with each other. These might be operating on behalf of two users trying to share information, or a person trying to gain access to the devices in a shared environment, or any of a number of other scenarios. When an agent in one society needs to invoke the services of an agent in another society, it first contacts a local resource manager. Note that even though the request is for an agent in a different society, the local agent still views the remote agent as merely another necessary resource, and delegates the task of finding the appropriate component to the local resource manager.

If the resource manager determines that the request involves the remote society, it then forwards the request to the Ambassador for the local society. In the Hyperglue implementation, this is handled by delegating the request to a class with the expertise to analyze resource requests and determine whether a request should be considered local or should be passed to a remote society for action. This decision is fairly straightforward – if the request is for a specific AgentID, and the society for that AgentID is remote, it is forwarded into Hyperglue. Otherwise, the decision is made based on the kind of agent being requested; if the request is for one of a set of agents that tend to live in spatial societies, the remote society is queried and used. Some examples of such "spatial agents" include:

- Device agents (which tend to be closely associated with hardware in an IE)
- The GuiManagerAgent, which is closely coupled with output devices
- Web browser agents, which tend to be shared applications, usually coupled closely to the operating system

The Ambassador agent acts as a point of contact for its society to other agent societies, and is registered upon startup with a global directory system called the Hyperglue Entity Directory (HED). Like real-world ambassadors, the Ambassador's function is to represent the agents in its society to other societies of agents. This includes sending out requests for handles of agents in other societies as well as receiving requests for handles to agents in its society.

When the local Ambassador receives the request, it consults the HED for the location of the Ambassador agent for the remote society. It then contacts the remote Ambassador and passes on the request. The remote Ambassador then passes the request on to its own resource manager, which makes the determination about how to fulfill the requirements (if at all), and passes back a handle for any found resources to the local side through the Ambassadors' connection, and finally back to the original requester.

One might wonder why the HED is not likely to become a centralized chokepoint in the system. Note, however, that the network traffic that the Ambassador must employ to communicate with other societies easily dominates the small traffic that is sent to the HED in performing the initial name lookup. As such, the HED is not likely to cause delays until the number of societies in the network becomes large. When that occurs, other approaches to societal and HED organization can be explored (see Section 4.6 for a description of one such approach.)

One of the nice features of this approach is that agents treat inter-society requests the same way that they treat requests within the society, easing agent programming greatly. The entire process of requesting remote agents is hidden behind the call to the local resource manager, so Hyperglue's inner workings are behind the "abstraction barrier" presented by the resource management framework. In addition, the inner workings of agent societies are encapsulated into the higher-level societies, easing the burdens put on the HED during the discovery phase.

In Metaglue, all resources are represented by agents[2]. For most physical resources in an IE, the agent that gets returned as part of a resource request is a device controller. This approach in Metaglue (and reflected in the base Hyperglue design) – making requests for APIs and getting agents in return – is a bit weak semantically, since it doesn't allow for the possibility of retrieving services on anything more than a strictly operational basis. That is to say, if it's not expressible as a service or API description, it can't be retrieved. Hyperglue addresses this through semantically richer descriptions and context-dependent requests, which

[2]To be perfectly accurate, there *do* exist means to return resources that are not agents in Metaglue – in theory, this would allow the reservation of resources that represent something other than an API-enabled service. One example might be an object representing and reserving the connection between a computer and a display made through a video multiplexer. However, most resource managers in Metaglue neither recognize nor return such non-agent resources, so I ignore them in this document.

are described further in Chapter 5.

4.4.2 HED Lookup

The HED mechanism itself can be implemented through a number of different mechanisms, since its functionality is little more than a "yellow pages" reference, mapping a society name to the location of an Ambassador agent. Several such systems exist, including the Intentional Naming System[2], the IETF Service Location Protocol [42], or even simple additions to the venerable DNS[13].

There are currently two different implementations of the HED lookup mechanism. The simplest, used for testbed platforms of Hyperglue, is to use Java's RMI to store the locations of Ambassadors in a known registry, keyed by the society name. Although simple to use, this natually generates scalability and robustness issues, since the RMI registry needs to be in a known location and constantly in operation in order to function properly, and is thus likely to be insufficient for a long-running IE implementation.

Another version of the HED has been implemented over the Intentional Naming System. In this version, an INS name-specifier is created specifying the society name and contact information for the society's Ambassador. When a society starts up, it is registered into a nearby Intentional Name Resolver (INR), and that information is broadcast to other INRs in the network as needed. Then, when an HED request for a society is made, the nearest INR is contacted to locate the Ambassadors that match the given society name.

As mentioned before, and as pointed out in Lilley [24], there are scalability issues with INS when the number of entries grows too large. Those same scalability issues effect the INS-based version of the HED as well, but by limiting the number of records stored in INS and only using it for storing location information (rather than for every service in the world), we avoid the issues that impact INS' ability to handle large-scale environments.

4.5 Hyperglue and the Publish-Subscribe Model

Publish-subscribe ("pub-sub") mechanisms like the NotifierAgent, in which a single clearinghouse receives subscription requests and also forwards publication requests to its subscribers, present a minor problem in the Hyperglue model. As mentioned previously, Metaglue agents make extensive use of the NotifierAgent, to allow for easy creation of server-client architecture designs, and also to create mobile user interface constructs, where the details of the user interface presentation is independent of the workings of the agent applications. This is also used for creating event-driven agents, which use the publications to determine changes in the world and update their internal state accordingly.

Separating the societies presents some issues with systems such as the Notifier, however. In short, we need some of these notifications to cross the societal barrier. As an example, consider Metaglue's use of the Notifier in user interfaces using the GuiManagerAgent, say, for a simple chat application. The Metaglue design separates the control logic (the component that receives chat messages and processes them) from the interface and display components, and uses the Notifier to pass messages from the control logic to the interface. Although the controller certainly lives in the user's society (since the user should maintain control over his username and password for the chat system), the display that he's using to show the information may not – in a shared environment, the display is owned by the space, not the user. Therefore, either we allow the interface to be exported to the space's GuiManagerAgent, or we disallow this kind of usage. To maintain some level of usability, the publish-subscribe system must allow some notifications to cross the societal barrier from the user to the space.

With a design like the society model, one can envision the publish-subscribe system being handled in one of two ways. Either a global society is used to serve as a clearinghouse for all subscription requests in the system (totally ignoring the boundaries of societies), or each individual society can use its own clearinghouse to handle local message traffic. Hyperglue's design chooses the second option, in an attempt to limit message congestion.

With a global design, the central society would easily become overloaded with messages that most agents in the world don't care about. For example, an environment running a music application might broadcast information about the currently-playing song to the users within the environment, but users outside of that environment have no interest in this information. Sending all messages to a central dispatcher, when the vast majority of subscribers are uninterested in the content, could easily swamp the dispatcher. The dispatcher would need to be carefully designed, both to ensure that messages are quickly sorted and sent, and that messages aren't being sent to uninterested parties.

In contrast, the Hyperglue approach is for each society to provide its own pub-sub mechanism internally. Although this does make the number of messages manageable, and lowers the need to make each dispatcher a high-performance one, we are left with the problem of transferring messages outside of the current society. Turning again to the music application design, when the environment wishes to broadcast a change, it cannot limit the messages solely to the environment's society; it needs to also broadcast the messages to the users' societies so that they can handle the information.

To accomplish this, we add another level of indirection to the pub-sub design. In Hyperglue, requests for a Notifier are instead replaced with requests to a HyperNotifierAgent, which (in addition to performing the usual duties of the Notifier) forwards subscription requests to the society for the current location, and also forwards local notifications to requestors from other societies. As with the other aspects of Hyperglue's design, this distributed layout allows societies to communicate with other societies without forcing agent programmers to differentiate between intrasocietal and extrasocietal communication.

In order to differentiate between notifications that cross the societal barrier and those that shouldn't, and thus avoid infinite loops as messages are sent back and forth between socities ad infinitum, notifications (called "secrets" in Metaglue parlance) are first wrapped up in special HyperNotifierSecret objects, and then for-

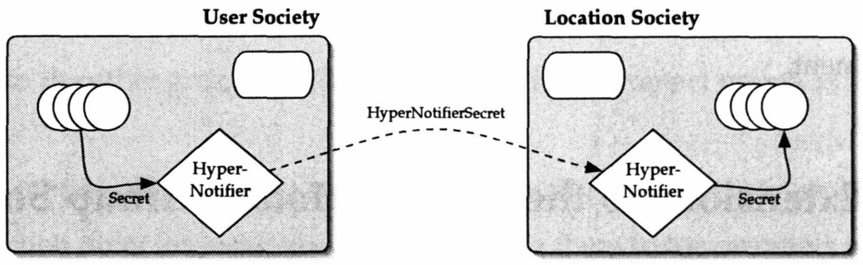


Figure 4-4: Publishing using the HyperNotifier

warded across to the remote society (Figure 4-4). When the HyperNotifierAgent receives a HyperNotifierSecret object, it only allows it to work locally, and will not re-forward it to another society.

4.5.1 Improvements to Metaglue: RMI Classloading

Some improvements had to be made to the base Metaglue system in order to facilitate the implementation of Hyperglue. One such improvement was the introduction of RMI-based classloading.

Java's RMI design requires that both sides of the communication have access to the class structure for the remote object. This is due to the fact that the RMI stub implements the same interface that the remote object does, and therefore all of the remote object's API information, including method names, parameters, and return types, must be known to both the local stub and the remote object as well. Because Metaglue is based on Java RMI, this means that under normal circumstances, all agent-to-agent communication would require that the codebase be shared, so that all interface information is equally available to both sides of the connection.

However, this is problematic for any long-running agent system, where a user may wish to introduce a new agent into an existing environment without having to restart the IE. To solve this problem, Metaglue extends Java's classloading facility, and automatically queries all Metaglue platforms when it needs information for an unknown class. It then loads the class definitions – for both new agents and subsidiary objects – over the RMI connection at runtime. This allows a user to write a new agent and introduce it into an existing IE without restarting the entire environment.

4.6 Extensions to the Society Model: Group Societies and Ad Hoc Communication

In section 4.2, we discussed a design for agent societies in which each society represented either a person or a place (usually an IE). Unfortunately, such a puritanical division leaves gaps in the agent model.

For example, consider an application where a user wishes to broadcast a message to the members of a workgroup. If the user is a member of the group, then he can probably send the message directly. However, there are many cases where the user would either not know all the members of the group (because he is an outsider), or where the workgroup may wish to place limits or filters onto messages being sent to the membership.

One such example can be found by looking to our third scenario, in which Ellie is sending a message advertising free donuts to the graduate students in the lab. In this scenario, there exist a number of research groups which are geographically associated with the lounge, each of which contains a number of graduate students[3]. These groups are not necessarily associated with any given room, so we cannot simply deliver messages to the grad students associated with a space. Given these conditions, it makes a great deal of sense to note which research groups are associated with the lounge, and contact those groups to locate their students who are available for the message.

In these cases, it is useful to have the same kind of divisions that the society model sets up between users and places, but instead acting as a representative for the workgroup. Because of this, we introduce the notion of a "group society," which acts as a collection point for a number of other societies, and whose ambas-

[3]The notion of "geographically associated" is one that would be provided by a semantic data store that can describe such relationships. This is described further in Chapter 6.

sador can act as the point of contact for the entire group, forwarding messages and requests to the other group members or to designated contact points as appropriate.

In conception, this is similar to how modern mailing list software works, acting as a collection point for messages and dispatching them to the members of the list, or to other sublists as required.

The group society can also be used to create ad hoc collections. These are useful when two societies may need to collaborate on a task, but don't otherwise have a common, pre-existing group in which they can claim membership. Such an ad hoc society would likely only exist for the duration of the task at hand, but can allow the participants to disengage or re-engage freely until the task is complete. One example is our "chat" application, in which two or more users may wish to come together for a conversation – although one user might wish to create such a chatroom, the creating user doesn't "own" the conversation in any reasonable sense. As such, the central application for the chat room would best exist in a temporary society, with the proviso that when a certain agent stops running, the society can be dismantled and any allocated resources released.

Because they have knowledge of their child societies as well as their location in a parent society, group societies can also serve as nodes in a hierarchical layout for the HED service. In short, ambassadors trying to find a given society would first examine an HED node attached to the nearest encompassing group, which could then return either a local society or direct the request elsewhere to return the information to the requesting ambassador.[4]

4.7 Revisiting the Scenarios

Let us briefly turn back to the three scenarios and see how Hyperglue and the societal model addresses the necessary infrastructure requirements, and explore

[4]Such a hierarchical, group society-based version of the HED has not yet been implemented. Such a design, however, could prove to be a fairly flexible, practical addition to Hyperglue, and so is presented here as a signpost for future work.

where the knowledge needed for each task lives.

4.7.1 Messaging

Here the use of the society model is fairly clear – each person (Alan, Beth, Charlie, Diane) has a separate society of agents. The agent handling Alan's message delivery attempts to locate the message delivery services for each recipient. Alan's resource manager will determine that these agents live in a separate society, so it will forward the requests for those agents to Alan's Ambassador.

Alan's Ambassador uses the HED to locate the Ambassadors for Beth, Charlie and Diane, and then forwards to them the request for a message delivery service. Because Beth, Charlie and Diane are in situations where notification through personal services (e-mail, SMS, personal laptop display) is appropriate, agents within their societies will be able to satisfy the service, and stubs for communicating with them will be returned to Alan's original requestor.

If Beth were in a situation where she would make use of her environment's facilities to deliver the message (over the room's speakers, for example), the message delivery service she used would in turn contact the room's Ambassador and open a new request for a message delivery service from the room's society. Once it receives it, it passes on the message text to the room's service, so that the text is delivered appropriately.

To break this last case down step-by-step, assuming that Beth's preferences are to display the information on a local display in Beth's current location.

1. One of Alan's agents receives a request to deliver a text message to the Beth society. It builds a service request based on the society requested and the message delivery API, and requests it from Alan's resource manager.

2. Alan's resource manager knows that "Beth" is a reference to a different society, so it forwards the request for an agent to Alan's Ambassador.

3. Alan's Ambassador uses the HED to locate the Ambassador for Beth, and

forwards the request for a message delivery service on to her.

4. Beth's Ambassador uses the local service mapping engine to request the service.

5. Beth's service mapping system has a number of choices for message delivery, based on her preferences and current contextual situation. It gathers this information by consulting a local semantic storage that contains information about Beth (see Chapter 6). After it gathers context, the service mapping system can choose the best agent to satisfy the request (see Chapter 5).

6. Beth's service mapping system returns a handle to the agent to Beth's ambassador.

7. Beth's ambassador passes on the handle to Alan's ambassador

8. Alan's ambassador passes the handle to the resource manager, which hands it off to the original agent making the request.

9. Alan's agent uses the message delivery API to send the text to Beth's agent.

10. If Beth's chosen delivery agent doesn't need to know about devices in her current location, then it can simply act to deliver the message (say, through email). However, in this case, it wishes to open a display to view the information. Therefore, it requests a textual display from Beth's service manager.

11. Beth's service manager recognizes that it does not have a textual display in its society, and decides to try a local display instead. It thus consults Beth's location information to find out where she is, and redirects the request for a textual display to Beth's Ambassador.

12. Beth's Ambassador passes the request on to the room's Ambassador, which will act very much the same way as above, using the preferences defined for the room to determine an appropriate display to use, and returning a display service to Beth's delivery agent through the ambassadors.

13. Beth's delivery agent uses the display service to show the information.

Note how the information is compartmentalized: information about Beth's preferences and location is only consulted by Beth's agents, and thus only needs to be stored local to Beth. The only piece of information about Beth that Alan's agents need is the information needed to look up Beth's society, but nothing personal about Beth herself. Similarly, information about the room's preferences can be kept private to the room.

4.7.2 Application Mobility

Alan wishes to move the practice talk from his office to a different conference room, which might make use of different services. When he makes that decision, his practice-talk application has already retrieved services from his office's society, so that he has access to agents that are displaying his slides and the timer.

These agents make use of the GuiManagerAgent, so that they decouple the actual display from the control logic that decides which slide to show and when to advance them. As such, Alan's society is running the application agent for the practice talk, as well as the control agents for the slides and for the timer. His office's society is running the GuiManagerAgent which is displaying the user interfaces. As mentioned in Section 4.5, this system uses the publish-subscribe architecture of the HyperNotifierAgent to send GUI control messages between Alan's agents in his society and the GuiManagerAgent in his office's society.

When Alan decides to move the talk into the conference room, his agents only need to contact the conference room's society for the appropriate GuiManager-Agent services that can handle a large screen layout for the slides and a smaller screen for the timer. Once it receives these services, the agents in Alan's office are asked to stop displaying the user interfaces and the agents in the conference room given the interface objects for display. No change needs to be made to the running interface agents; they will still be using the HyperNotifierAgent to send messages to the UI, and once Alan is in a new environment the notifications are redirected to the new society.

When Alan moves, his practice talk application will also check to see if the new location has recording services available. Since the conference room has permanently-mounted cameras, it contacts the conference room's society and is able to obtain services that can capture a record of the environment.

4.7.3 Free Donuts

As noted in the previous section, because Ellie does not specifically know which grad students need to be contacted, her message delivery application here needs to make use of group societies for the individual workgroups associated with the lounge. Her message delivery application would contact the group societies for message services, indicating that the message is low-priority and only for those people who can receive the message soon. The individual group's message services then determine which of their members satisfy such requirements, and send the message on to the appropriate people, using the techniques described by Alan's message delivery application.

The group societies also do a good job of compartmentalizing the information about the group. Ellie's agent can merely send a request specifying some message parameters to the group society, and does not need to maintain any knowledge about the individual members of the group.

Still undefined in all of these scenarios is precisely how we select the proper service given a request, and how we query the knowledgebase to determine a room's status or a user's location. These aspects are defined further in subsequent chapters.

Chapter 5

Service Mapping

The primary goal of the Hyperglue design is to provide infrastructure to support a simple principle – that decisions should be made by those affected by the outcome of the decisions, and that no element of the system should be able to directly control resources that belong to another.

We have seen in the previous chapters how Hyperglue goes beyond most existing pervasive computing environments in creating first-class representations (i.e. societies) of individuals and spaces. It establishes a protocol in which information is encapsulated within these societies, making it necessary for agents in one society to interact with agents in another society indirectly. Such interactions are mediated by the Ambassador agent of each society, preventing agents representing one entity in the real world from appropriating resources that belong to another.

In addition to this basic principle, Hyperglue also aspires to provide a high level of adaptability to variations in the environment. Because Hyperglue societies are intended to be long-lived entities, they must be capable of responding adaptively to new situations in the environment – for example, to the loss of a resource or to the acquisition of a new resource. In addition, because the interactions that Hyperglue is intended to support mirror those of normal human society, Hyperglue components must be able to adapt to different user preferences.

In this chapter, I will describe how the Hyperglue framework achieves these goals by the use of:

- A *decision theoretic* approach to decision making.
- A *focus on high-level goals* rather than on resources and procedures.
- The use of a *library of alternative plans* for each known goal type
- A *plan monitoring system* that traces the execution of plans, detects and diagnoses breakdowns and recovers as appropriate[1].

5.1 Overview

Before jumping into the details of the mechanisms that I have designed for Hyperglue, it would be useful to see the decision-making architecture in broad overview[2]. In the rest of this section I will describe each of the major components of this architecture including the rationale for each choice. In the following sections, I will then describe the specific mechanisms that are implemented in the current Hyperglue system. After that I will briefly describe additional components of the Hyperglue decision-making architecture that are under development (often as joint work with other members of the group) but not yet incorporated into the operational prototype.

5.1.1 Services

In contrast to most current pervasive computing environments (with the notable exception of O2S[32]), we place our emphasis on abstract requests for service, rather than on the discovery of specific resources. The rationale for this is to allow the Hyperglue system to adapt to changing conditions while providing the user the best possible service available. Consider even a simple user goal such as providing a reasonable level of lighting. Rather than requesting control of a specific lamp, which might be broken, a Hyperglue user (or an agent within the Hyperglue

[1]This element of Hyperglue was designed and implemented by Gary Look.
[2]Much of the design in this chapter is joint work with Howard Shrobe.

environment) makes a request for an "illumination service". This allows the system to find an alternative method for illuminating the space; for example, it might open the drapes or use a different lamp.

5.1.2 Plan Library

Requests are intended to be abstract enough to allow more than one way of providing the service. Hyperglue societies provide a library of methods capable of providing each service that a user may require. Accompanying each plan is a rich set of meta-data, encompassing:

- what resources the plan will require
- what constraints apply across the ensemble of resources; and
- how the quality of service provided will depend on those resources.

A complex plan is also accompanied by its decomposition into sub-tasks and the flow of data between these sub-tasks. Each sub-tasks is either a primitive task, with directly executable code, or is itself a service request for which a sub-plan must be fetched.

5.1.3 Service Quality Parameters

Because services are intended to be rather abstract, there is a range of "quality of service" that will result from different methods of rendering the service. For example, opening the drapes will provide *natural* lighting, while using an incandescent bulb will result in *artificial* light. Opening the drapes is likely to reduce *privacy*, while using artificial lighting sources won't. Different users will have different preferences over these service qualities and will make different trade-offs. One user might value privacy over natural lighting while another might make the exact opposite choice. Thus, service qualities must be explicitly represented and reasoned about dynamically, taking into account the quality parameters and the user's desires.

5.1.4 Preferences

The assumption made in Hyperglue is that the user does not have a direct interest in the details of how a service is rendered; she cares only about the outcome of the process. For this assumption to be tenable, we include within the notion of "outcome" a set of preferences stating the user's tradeoffs between the different qualities of service that might be delivered. We allow these to express degree of preference and to apply to arbitrary combinations of service parameters; for example, to continue with the lighting example, the user might state that intense illumination is twice as good as low illumination, or that the combination of intense illumination and strong privacy is much better than natural lighting alone.

5.1.5 Utility Functions

Because there are trade-offs to be affected, it is useful to transform these statements of preference into numerical values that can be more easily compared. In addition, as we will describe more fully, there are also trade-offs to be made in terms of which resources are to be used. In order to allow all of these trade-offs to be dealt with uniformly, we take advantage of a graph-theoretic algorithm (based on work by McGeachie and Doyle) for converting the user's preference rankings into a linear ordering over sets of service quality parameters. The algorithm constructs a graph whose nodes are combinations of service quality parameters and whose weighted arcs are the user's expressed preferences. As long as the preferences are consistent, the graph will be acyclic and simple dynamic programming techniques can be used to deduce a total numerical ordering consistent with the preferences. This numerical ordering is scaled to range between zero and a user-supplied parameter that reflects the maximum cost the user would be willing to expend in order to reach the optimum quality of service.

5.1.6 Context

An individual's preferences are not constant over all times. To continue the lighting example, it is obvious that the task the user is conducting (e.g. reading or watching a movie) or the user's location may greatly affect her preferences for the intensity of lighting and the naturalness of the spectrum. Although there are certainly many other dimensions to the notion of context (e.g. the user's emotional state), Hyperglue's model of context is limited to a few dimensions:

- the time of day
- the task being conducted when the service request is made, and
- the location of the user.

In addition, Hyperglue maintains a context stack representing the nesting of subtasks within larger tasks. The primary role of the context is to provide a set of default preferences over the qualities of the service to be rendered. The user is free to override such choices, but our hope is that the default choices are almost always reasonable and will be adjusted only rarely, as part of a dialogue between Hyperglue and the user. In summary, context determines preference, and preferences are used to rank different methods for rendering a service (via the derived utility function).

5.1.7 Resource Cost and Allocation

Different methods will require different types of resources, but not all resources are equal. Some are more valuable, others are more available. Some resources deplete consumables (e.g. paper in a copier) while most resources have a depreciation cost (e.g. the bulb in a projector). In order to reflect these differences, Hyperglue places a cost on the use of a resource. Ideally, this cost would reflect the integration of all such factors, but the current Hyperglue model is relatively weak. Hyperglue attaches a fixed cost to the use of each resource, reflecting the relative value of that resource. In addition, when there is a class of equivalent resources (e.g. more than one projector), the cost reflects the intrinsic value of a resource in the class as well as the size of the available pool. When a method is selected and resources are allocated to that method. Hyperglue's resource manager updates both its reservation tables and the cost estimate. Similarly, when a method completes execution and returns a resource, the reservation of the resource is eliminated and the cost is re-estimated. The motivation for applying costs to resources is to allow Hyperglue to be cognizant not only of the benefit that a method will deliver to a user, but also to recognize that this benefit does not come for free and that the two must be traded off against one another.

5.1.8 Decision Theoretic Choice

These elements are all combined into an overall decision theoretic algorithm. This algorithm examines each method capable of rendering the service and then for each such method finds all possible sets of currently available resources that are consistent with the constraints of that method. For each such combination of method and resources, the cost of the resources is calculated. In addition, the quality of service parameters are calculated and ranked by the utility function derived from the preferences leading to a numerical benefit. Finally the difference is between resource cost and benefit delivered is calculated and the combination which maximizes this net benefit is selected.

5.1.9 Interactions Between Societies

It is my claim that this approach allows Hyperglue to adapt to a broad spectrum of varying conditions while providing the best possible service. So far, the discussion has dealt only with the decision making within an individual society. However the extension to inter-society requests is relatively straightforward: The requesting society forms a service request including the set of preferences; the receiving society adds to this additional preference rankings reflecting its interests. If these joint preference rankings are consistent, things proceed as above, with the receiving society selecting a set of its resources and a method that optimizes the cost-benefit trade-off. If the preferences are inconsistent, then the requesting society is asked to drop at least one preference involved in the inconsistency and the process iterates. Notice that this preserves the principle that each society maintains control over its own resources while still providing the best service to the requestor that is consistent with the preferences of the society that owns the resources.

In the remainder of this chapter, I will first fill in many of the details omitted in this overview discussion. I will then illustrate how Hyperglue's decision making architecture supports our scenarios. Next I will review some of the short-comings of the current implementation. Finally, I will conclude with a discussion of other aspects of the Hyperglue decision making architecture (e.g. plan execution monitoring, access control, diagnosis and recovery, event driven reactive processing).

5.2 Mapping User Preferences to Utility Functions

The reason for using preferences to drive service requests is simple – with preferences, users don't need to know precisely what contextual situations they are planning for, or provide guidance for each possible contextual situation. The alternative can get aggravating, because service selections may be highly dependent on context, and the resulting ramifications can become bewildering. For example, consider the simple case of deciding which display to use in an office for text output:

- Normally, an important message should be sent over email.
- ...except when the primary user is in the room, in which case it should be sent over the loudspeakers instead.
- ...except when the user is on the phone or a meeting, in which case it's better to send it to the main display on the desk.
- ...except when the projector is being used, in which case it's better to put the item there.

- ...unless the message is considered private, in which case putting it on the projector that's visible from the hallway is a bad idea...

And the list goes on, with exceptional clauses providing more and more context that needs to be considered, and perhaps overloading the network with conditional values.

So, instead of focusing on *how* to do the task, we switch our focus to specifying the desired outcome (e.g. the goal and the quality of service) and allow the Hyperglue system to dynamically determine the best possible approach in the actual context of execution.

Most services can be categorized based on different 'abstract' parameters, such as quality, speed, privacy, etc. Preferences are specified by providing a partial ordering based on the values of these parameters. Not only are there orderings based on the single parameter ("I prefer high privacy to low privacy"), but they can specify orderings across the parameters, so that the user can define that they prefer output quality to speed. We also can allow weighting on the preferences, to model a strong preference as compared to a weak one.

Preference matching is inherently easier for the user to handle, because they no longer have to specify a series of more and more complex contextual conditions in order for things to work properly. Instead, the user specifies preferences with statements such as "All other things being equal, I care more about strong privacy than fast response times." The advantages here is that the user doesn't have to know the space as intimately, and these preferences are applicable in IEs that the user has never seen.

However, to avoid overwhelming the user (and to allow for new parameters that get added by new devices), it's desirable to specify partial ordering for only some of the parameters, and allow the system to determine the appropriate comparisons based on these partial orderings.

We accomplish this by first taking the user's preferences and turning them into *utility functions*, which can return a numeric value for a service based on the user's preferences. The inputs to the function are the qualities of service that an indi-

```
(define-goal (illuminate)
  (privacy private public)
  (source natural artificial)
  (illumination bright moderate soft))
```

Figure 5-1: Lisp notation defining a simple illumination goal.

```
'((source natural (>> 2) source artificial)
  (illumination moderate (>> 3) illumination bright)
  (source natural (>> 5) privacy private))
```

Figure 5-2: Simple code describing a preference. Here, the user is indicating that natural light is preferred to artificial, illumination is better in moderation, and natural light is much more important than keeping things private.

vidual service provides, and the utility function can thus be used to compare and contrast services based on the user's preference set.

5.2.1 Utility Function Calculation

One key element in this process is to transform a set of user preferences over the service qualities into a single numerical utility function. Consider the (admittedly simple) goal of illuminating a room. The service qualities we include are: whether the lighting is natural or artificial, whether the lighting is bright, moderate or soft and whether privacy is maintained (since opening the shades impacts privacy). In our system, this information is conveyed in a Lisp-like notation (see Figure 5-1).

The user's overall preferences are expressed as a collection of individual statements, each of which compares one set of features to another. For example, an individual quality preference statement might state that natural light and privacy is twice as good as bright artificial light, "all other things being equal." The user's overall preferences are then a set of such individual preferences.

For example, the code in Figure 5-2 would represent the fact that the user prefers natural light twice as much as artificial, moderate intensity three times as much as a bright intensity, and natural lighting five times as much as privacy.

To convert a set of service quality preferences into a utility function we can

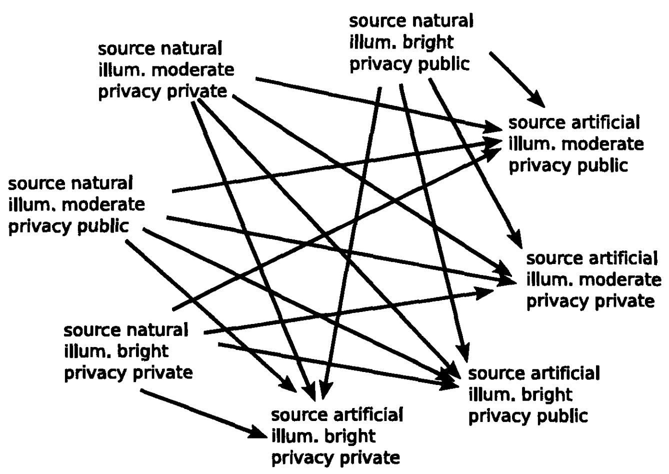

source natural illum. bright privacy public
source natural illum. moderate privacy private
source artificial illum. moderate privacy public
source natural illum. moderate privacy public
source artificial illum. moderate privacy private
source natural illum. bright privacy private
source artificial illum. bright privacy public
source artificial illum. bright privacy private

Figure 5-3: Utility Function generation: Creating nodes (here using three binary parameters) and drawing lines which correspond to preferences ("natural light is better than artificial").

rely on techniques developed by McGeachie and Doyle [26]. The essential idea is to transform the assertions above into a graph, in which each node of the graph represents a particular assignment of values to the preference variables. Then an arc is drawn between any two nodes that are related by a preference statement (see Figures 5-3 and 5-4). After all arcs are added, we then assign a value to each node by first assigning the value '1' to any node with no outgoing arcs. For the other nodes, the value is found by iterating over all outgoing arcs and maximizing the product of the arc weight and the value of the target node of the arc (Figure 5-5). The utility function then simply looks up the node corresponding to a set of preference variables and returns that node's value.

In the event that the feature set of the service is under-specified – for example, it provides a value for "source" but not "privacy," – the algorithm chooses the node with the minimal value for utility. We choose the minimum to avoid dominating services that are more closely specified to give higher utility values (see Section 5.4 for better ways of handling this condition, however).

The McGeachie-Doyle algorithm used boolean-valued parameters; therefore, its representation of a node in the graph is a bit-vector with each bit representing one parameter. One of the drawbacks of this approach is that parameters can only

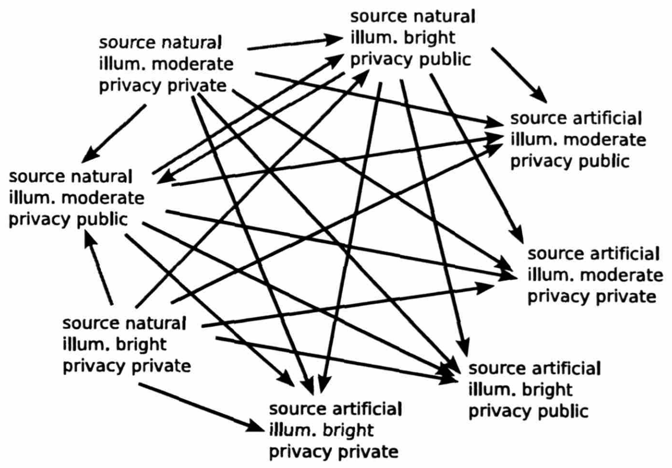

source natural illum. moderate privacy private
source natural illum. bright privacy public
source artificial illum. moderate privacy public
source natural illum. moderate privacy public
source artificial illum. moderate privacy private
source natural illum. bright privacy private
source artificial illum. bright privacy public
source artificial illum. bright privacy private

Figure 5-4: Utility Function generation: Adding another preference rule ("natural light is more important than private privacy").

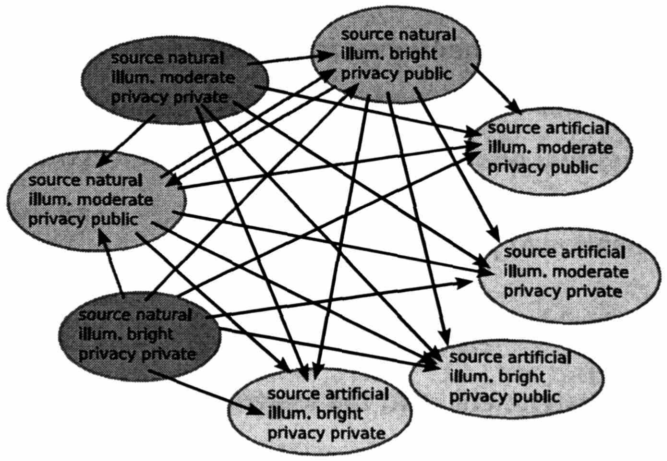

source natural illum. moderate privacy private
source natural illum. bright privacy public
source artificial illum. moderate privacy public
source natural illum. moderate privacy public
source artificial illum. moderate privacy private
source natural illum. bright privacy private
source artificial illum. bright privacy public
source artificial illum. bright privacy private

Figure 5-5: Utility Function generation: Assigning values to nodes based on their position in the network.

hold binary values – speed can only be "fast" or "slow", quality only "good" or "bad". We have extended this system to provide for multivariate parameters simply by turning a series of possible values (e.g. our illumination parameter that can be "soft, "moderate", or "bright") into a set of boolean values that can be fed into the utility calculation. An implementation of the algorithm can be found as an appendix (Appendix A.1).

In this modified algorithm, each parameter may take on one of a finite set of values. We therefore structure our nodes as a vector of (unequal sized) bit segments. Each segment has enough bits to encode each possible value of the enumeration – as such, its size is $\lceil \log_2 |E| \rceil$, where E is the set of values in the enumeration. The graph nodes are then represented as a bit-vector that is the concatenation of these individual segments. For example, our illumination goal has two parameters ("privacy" and "source") with two possible values ("public" vs. "private", and "natural" vs. "artificial") and one parameter ("illumination") with three values ("soft", "moderate", "bright"). Therefore we allocate one bit for each of the first two and two bits for the last, making a vector of five bits in total.

As in the McGeachie-Doyle algorithm, each preference generates a set of arcs in the graph (because each preference is stated "everything else being equal"). Thus, if the "foo" parameter is not mentioned in an individual preference statement, the interpretation is that this statement is true for any value that foo might take on. Thus given an individual preference statement we perform the following steps:

1. Find all unmentioned parameters;
2. generate all possible assignments of values to all of these parameters;
3. for each such assignment, add the assignment to both sides of the original preference statement;
4. find the nodes corresponding to the expanded left side and the expanded right side of this statement; and
5. add an arc between these nodes.

It is worth noting that this algorithm, like the original McGeachie-Doyle algorithm has a runtime that grows exponentially with the number of parameters. For this reason, and to avoid overburdening users with numerous options for preferences, keeping the number of parameters small is important. And indeed, in practice, the number of parameters seems to be bounded by the users' desire for simplicity.

There is one further modification of the original algorithm. A preference stating that I prefer "moderate illumination" to "strong privacy" should be interpreted as: "I prefer the combination of moderate illumination and *not* (strong privacy) to the combination of strong privacy and *not* (moderate illumination)." In the original McGeachie-Doyle algorithm, parameters take on boolean values; therefore, one simply adds to the left side the negation of the right and vice versa. If either side is a conjunction, then this results in a disjunction and boolean canonicalization must be performed, yielding more than one derived statement.

In our case, with parameters taking on a value from an enumeration set, the negation means that we must consider all other members of the enumeration set. This is easily understood via an example. Suppose we have the following preference:

```
(illumination moderate) (>> 2) (privacy public)
```

this is expanded to

```
(illumination moderate) and (not (privacy public) (>> 2)
      (privacy public) and (not (illumination moderate))
```

which in turn is expanded to:

```
(illumination moderate) and (or (privacy private))
  (>> 2)
    (privacy public) and (or (illumination bright)
                             (illumination soft))
```

Next, each side is put into the disjunctive normal form:

```
(illumination moderate) and (privacy private) (>> 2)
(or (illumination bright) and (privacy public)
    (illumination soft)   and (privacy public))
```

and then the "cross-product" of the two sides is formed yielding two preferences in this case:

```
(illumination moderate) and (privacy private) (>> 2)
(illumination bright) and (privacy public)

(illumination moderate) and (privacy private) (>> 2)
(illumination soft) and (privacy public)
```

Each of these preferences is then further expanded by adding in all combinations of the unspecified parameters as explained above, leading to an even larger set of arcs to be added to the graph. Of course, the last step before actually adding an arc is in converting from the "property list" format used above into the internal bit-vector position. This is done via simple book-keeping code that maintains a mapping between each parameter and its position in the bit-vector and the numerical value assigned to each symbolic value of that parameter.

5.2.2 Simplifying the Calculations Using Context

In the McGeachie-Doyle approach, the contextual elements of the system were presented as additional parameters to the calculation, allowing the user to specify that, for example, having natural light is preferable in the morning, but artificial light preferable in the afternoon, by creating combinations of service qualities with a "time of day" context parameter (see Figure 5-6).

Using the contextual parameters in this manner can create, as one might imagine, a rather large set of contextual parameters. Since, as noted earlier, the algorithm builds a graph with one node for each parameter combination, the graph

```
'((source natural time-of-day morning (>> 2)
                    source artificial time-of-day morning)
  (source artifical time-of-day afternoon (>> 2)
                    source natural time-of-day afternoon))
```

Figure 5-6: Specifying context in a preference.

size will grow exponentially with the number of parameters, and using context in this way could thus significantly increase the time required to generate a utility function.

One means of simplifying this situation, given the limited number of contextual elements that we are exploring, is to allow the user to set up different preference sets, and use the current context to infer the set of preferences we would like to use. We explore this further in the next section.

5.3 Context and Service Mapping

For many multi-agent systems, service mapping can be handled simply by examining the service request, and determining an appropriate agent that will satisfy the request. Indeed, this is the model adopted by many systems, including OAA and Metaglue. This approach, however, has some shortcomings, and foremost among these is an inability to adjust to the *context* of the request.

"Context," as used here, can be a catchall for a wide variety of issues:

- The services available to fill a request cannot be assumed to be constant, especially in a long-lived system like an IE. New devices and applications can be installed or removed, or equipment may break down, and the system will need to adjust to these changes in the environment.

- The "best" service to provide may vary based on various parameters, such as a request's urgency (if I want a message delivered swiftly, using a Western Union telegram may not be the most expedient method). Some services may be available only during certain times, and thus alternate services should be used when appropriate.

- A user may also have preferences for a default service, and also wish to specify alternate services which will be used under certain contextual conditions.

Indeed, at one workshop on ubiquitous computing, there was a discussion of what the word context meant in ubicomp, and resulted in a tongue-in-cheek definition:

> "Context is anything that you don't want to model explicitly in your system."

Whereas this definition is perhaps a little too vague, it contains a grain of truth – essentially, the context of "X" is usually highly dependent on what "X" is, and often context is whatever isn't in "X". Context may very well be something that you don't want to specify explicitly in your service request, but nevertheless you want applications to recognize and take into account without being told.

There's also another side to this quote – the necessary contextual elements are often unplanned by the designers of the system; often, they arise through normal use, by users who want the world to adapt to the current conditions in ways that could not be foreseen. For these situations, "context" cannot be a rigid notion, but one that adapts to the current conditions.

The real issue, for our purposes, is to define what *relevant context* exists, and to make use of it. As such, we model context along a limited set of axes. We also recognize that the coupling of context to service requests is indirect via preferences – we choose a method because it offers speedy delivery; many contexts might dictate a preference for speed and many others wouldn't. Thus, preferences separate context from method choice.

5.3.1 Context in Hyperglue

As noted earlier, we have chosen to model for Hyperglue a small set of elements to represent the contextual elements in play for a service request – the task being conducted when the service request is made, the location of the user, and the time of day. We choose these elements because they are at once the easiest elements in terms of obtaining information programmatically, and also arguably the most important, since they cover the user's main thoughts of "what I'm doing," "where I'm doing it," and "when it's happening".

For each of these, we create a simple ontology for specifying what's occurring. Time of day is broken down into simple units such as "morning" or "afternoon"; the task and locations are defined by simple ontologies that define relationships and usage to make specifying preferences easier – for example, noting that giving a presentation is a kind of meeting activity, or that room 32-225 is a private office of a faculty member.

The "current task" is actually represented as a short stack of tasks, so that we can, if appropriate, delegate to the previous task context if the current task is not actually something that helps us choose the correct service. This is inspired by Ajay Kulkarni's work on defining and switching task contexts in an intelligent environment [20].

To utilize the current context, we infer the user's preferences for his service request using the current context as a conditional element of the inference. As noted in the previous section, rather than incorporate context as part of the user's service parameters, we instead use context to infer parameters that should be in the user's preference set.

We do this by allowing the user to specify different sets of preferences given contextual situations, and then using the context to locate the set of preferences that is "most appropriate" for the given situation. In many cases, however, the user might be making use of the inheritance properties of the task or location ontologies, or not indicating that the value of a particular context parameter matters. With the prevalence of these inheritance properties and "don't-care" values, the means of searching for the correct set of preferences can be considered similar to that of generic or "multimethod" dispatch used in many object-oriented languages.

Multimethod dispatch – found in languages such as the Common Lisp Object System, Dylan, or Nice – is a means of choosing the best possible method based on the type of the arguments used in the request. Whereas in a conventional language, invoking a method specifies one and only one method can be chosen for a given name, a multimethod language will examine the types of all arguments and try to find one of a series of methods which most closely satisfies the arguments.

Similarly, a multimethod service selection engine will take in a number of arguments, including:

- the current location,
- the current task stack,
- the time of day, and
- the user making the request.

It will then use the types of the arguments and the ontologies that describe task and location hierarchies, and perform a lookup in a set of preferences to find the one that most closely satisfies the arguments. The resulting preference set is then compiled into a utility function to choose which of the available services best fits the current situation.

There are several means of implementing this type of selection. One simple means, similar to the system used in CLOS and used in Metaglue to perform method selection, is to examine all the possible matches for the given service, filter out all those that are completely incompatible with the current contextual arguments, and sort the remainder according to the specificity of the corresponding arguments – e.g., because 32-225 is a kind of "faculty office", a preference set designed for room 32-225 will be considered more specific than one designed for any faculty office. Once the possible matches have been sorted in this way, we use the highest-ranked set as the most appropriate for the given context.

5.3.2 Implementation Notes on the Context Selection

As mentioned, each element of context needs to be situated in an ontology. For this, we need to have agreed-upon ontologies for task types, location types, and even times of day or year. For design and testing purposes, we have used limited categorizations (time of day consists only of "morning", "afternoon", and "night", for example). Once the ontology is set, however, we have a set of location, temporal and task categories each of which forms a partial order. We note that services requests are categorized within the task ontology.

Contextualized preference mappings are then expressed by rules whose left-hand side includes the requested service and the various elements of the context (time, place and current task) and whose right hand side is a set of preferences. Each element of the left-hand-side is specified by class within the ontology; a rule applies if each element of the context is a member of the specified class (including inheritance) and if the requested service is an element of the task class specified in the rule. For example, consider a complex context-dependent preference rule:

```
if the requested task is an information-delivery task
   and the location is an office
   and the time is normal-working-hours
   and the current-activity is desk-work
then prefer intrusiveness low to speed fast by 2
       and prefer privacy to speed fast by 3
```

This rule would be relevant if the current location is 32-225, it's 4PM, the user is writing a paper and the requested service is to notify him of an upcoming seminar.

The inference process proceeds in 3 steps:

1. Fetch all applicable rules;

2. filter out dominated rules; and

3. combine the results

The first step is straightforward: given the actual values of the requested task and the contextual elements, we find all rules whose specified categories apply to the corresponding actual values.

The second step maps over these and eliminates any rule that is dominated by another. Rule A dominates Rule B if every argument of A is strictly more specific than the corresponding argument of Rule B. Given that the ontology forms a partial order, given any two values within it one may be more specific than the other or the two values may be incomparable. Two rules may be incomparable if 1) they have a single argument where the values in the two rules are incomparable; and 2) each has some number of arguments that dominates an argument from the other. At the moment we do not filter out incomparable rules and we include both in the final set. However, we believe that a better design would check for this at design time and notify the programmer that there is a conflict.

The final step runs all the results and combines the sets of preferences from each individual rule into a single set.

We describe this as a rule-based inference process, but in practice the implementation could employ capabilities from the underlying programming language. For example, given the multimethod and method combination capabilities of CLOS and other languages, this whole process is implemented just as method invocation.[3]

5.4 Extensions to the Service Mapping Algorithm

In Section 5.2, we have seen how a set of preferences defined by a user can generate a utility function, and how such a utility function can be used to drive the selection of an appropriate resource. Because we also have a resource cost associated with each provided service, the selection algorithm makes a choice that maximizes the difference between the calculated utility of a service and the service's aggregate resource cost (this difference is the "benefit" of a service). This code for this algorithm is in Appendix A.1.

It has been noted, however, that this design may not take into account a full

[3]To be precise, the method combination under a CLOS-style system is "append," and the task, place and time ontologies are mapped onto the type system of the language.

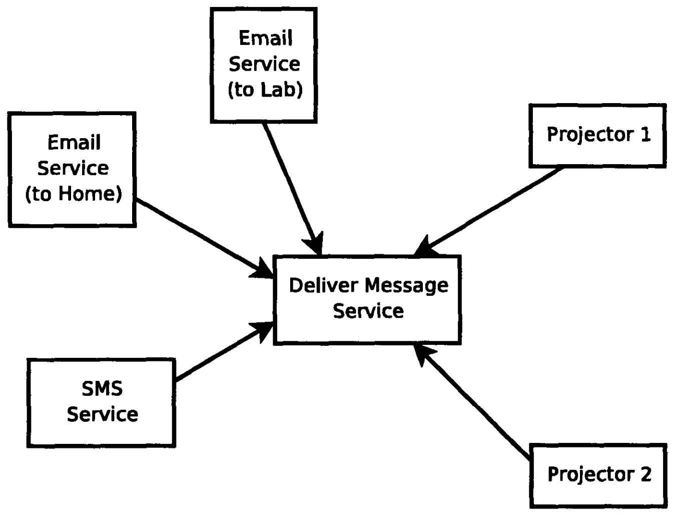


Figure 5-7: Evaluating all services at the top level.

decision tree for resources. For example, when deciding to display a message to a user, the algorithm as designed would be deciding between, say, delivering SMS text to a cell phone, sending email, or displaying the message on a nearby projector. Given the current algorithm, the most accurate way of taking into account all the possible parameters would be to consider all the possible means of delivering every possible service at the same time (see Figure 5-7), rather than as a hierarchical decomposition (Figure 5-8). This results, however, in a single global decision engine, and could generate scaling problems when the tree becomes overly large.

We could attempt to remedy this by examining the services hierarchically, using a generic "email service" or "projector service", which utilizes estimates of service quality and resource costs, based on the average quality and costs that its sub-services typically render. Then, the service mapper could be called upon again to choose between the individual projector or email provider. The problem with this, of course, is that we're not taking a true account of the current quality or cost of the individual services, so we may end up making a non-optimal decision. For example, if in general projectors give us a high utility in comparison to cost, using a multi-step algorithm will tend to choose them. But if a high-utility projector is unavailable, its resource cost will be prohibitively high, and we may end up

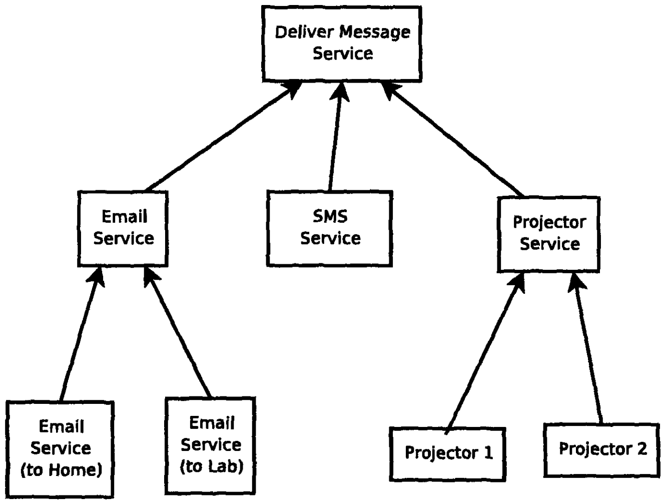


Figure 5-8: A hierarchical decomposition of services.

choosing a less optimal projector when sending email would be preferable. In effect, we would have chosen a local maximum for our utility calculation, and missed the global maximum.

In this rest of this section we describe an extension to the service mapping algorithm describe so far. This extended service mapping engine supports: 1) hierarchical task network planning; 2) accurate decision-theoretic selection of the set of resources and method and (recursively methods and resources for the required sub-services); and 3) branch-and-bound pruning of the search tree.

5.4.1 Using Bounds Calculations

As it turns out, the algorithm can be easily modified to take into account a hierarchical decomposition of services, and perform task planning in that case far more correctly. First, we modify the algorithm so that the generated utility function is no longer a simple mapping of service parameters to a single numeric value. Instead, we create a function that describes the *range* of utility that the function provides. Hence, the generated utility function is no longer a simple mapping of service parameters to a single numeric value, but instead takes any subset of the service parameters and returns two numbers – the maximum and minimum utility values that can be realized given the specified parameters and any value of the unspecified parameters.

In the original algorithm calculating the utility of a given service, we simply found the node that exactly matched the specified qualities of service. This is done by iterating over the provided parameters, finding the corresponding numeric encoding for that value of the parameter and concatenating these into the overall bit-vector description of the parameters. This is then used to index into the table of all nodes in the space, thereby fetching the utility corresponding to the specified set of service parameters.

However, like the preferences, the service itself might be under-specified; it might provide a value for "quality" but not "privacy", for example. In effect, the service itself has parameters that are "don't care" values. Whereas the initial version of the algorithm simply chose the minimum utility value to avoid dominating better-specified services, we can instead examine the entire set of nodes that match the service parameters, and discover the minimum and maximum values for the utility calculation. This can be done with the following simple alteration to the `utility` function (shown in Appendix A.2):

- Initialize the maximum and minimum values to "unknown".
- Generate all combinations of all values of the unspecified parameters.
- For each such combination,
  - append these to the provided parameters;
  - fetch the corresponding node; and then
  - maximize and minimize this value into the maximum and minimum values respectively
- When all combinations are evaluated, return the maximum and minimum values.

Note that if all parameters are specified this returns the exact value as both the maximum and minimum.

5.4.2 Hierarchical Task Planning

Once the bounds calculations are in place, we can change service descriptions to better include sub-service requests (sub-goals, essentially). For example, the "projector display" service for a given room includes a request for a simpler "projector" service, and we use that service to get an API for performing the actual display operation. Also, we provide individual resources for "left projector" and "right projector," along with their appropriate resource costs. This creates a hierarchy of service descriptions which looks more like the design in Figure 5-8.

The design for the plan selector needs to change in order to take this into account. It now needs to descend the service hierarchy to locate the best solution. We do this in a depth-first traversal of the service tree.

In addition, we use unification – in the logic programming sense – to connect the service parameters of the sub-services explored lower in the tree to those of the top-level service request. This means that method descriptions need to specify how the choices of resources and methods for sub-goals affect the service parameters of the sub-service for which this method is being explored. Unification connects these parameters to those of the next higher level sub-service in the tree, and so on up to the top-level service request. When returning from a branch of the search tree, the bindings of the variables are undone, using standard logic programming mechanisms.

Luckily, the utility bounds calculation of the previous section allows us to prune away sections of the tree during the search:

- The utility function that is invoked on each step of the search needs to only examine the service parameters of the top-level request. In other words, we never need to bother with attempting to find a heuristic function for each sub-goal that might lead to maximizing the value of the solution to the top level goal. We simply evaluate each possible realization of a sub-goal in terms of its effect on the top-level goal.

- At each node in the hierarchy, we use the utility bounds calculation to calculate the range of utility that the service can provide. This provides a maximum bound on the utility of the service and its sub-services, and thus a maximum on the net benefit of the plans.

- As we descend the tree branches, we accumulate the resources that are required for the entire branch, and the cumulative cost that those resources impose. Thus, as we descend the branches, resource costs can only rise.
- Because of this, the benefit of a service (utility minus cost) can only decrease as we descend the tree and bind the resources associated with its subservices. Hence, if the maximum possible value for the utility of a branch drops below the currently calculated maximum value for the benefit, we can prune away the further descent of the tree at that point.

In effect, this is a depth-first branch-and-bound traversal of the tree. The pruning algorithm improves the runtime of the overall algorithm, so that we don't have to do an exhaustive search of the possible space.

Each method description includes a set of resource requirements and a set of sub-services. The algorithm begins with the original service request and then proceeds by depth first search, conducting the following steps on each sub-service request:

- Consider each method for the service.
- For each method, considers all possible set of resources matching the resource requirements of the method description.
- For each set of resources, calculate the added cost of these resources and add these costs to the total resource cost accumulated so far.
- Use the method description to bind any service parameters effected by the choice of resources.
- Calculate the maximum of utility possible given the set of specified top-level service parameters.

- Calculate the difference between the maximum utility and the total resource cost accumulated so far and consider this as an estimate of the net-benefit of this branch of the search tree.
- If a complete solution has already been found compare its net benefit to the estimated net benefit of the previous step. If the estimated net benefit is less, cut off this branch.
- Otherwise, consider each sub-service request in the method description and recurse. As each sub-service is explored, its effect on the top level service parameters is calculated and the bounds on realizable utility are recalculated.
- When a complete solution is found, calculate its net benefit (utility minus resource cost). If this is larger than the best net benefit found so far, update the best net benefit to the new value.

The critical step in this process is the cut off. To see that the cut off is admissible we need to consider the following two points:

1. The total cost of the resources used on a search branch only increases as the depth first search goes deeper into the tree. This is true because each sub-goal only adds new resources and no resources have negative cost.
2. The maximum realizable utility can only decrease as the search goes deeper in the search tree. This is more subtle, but still easy to see:

   As the search proceeds down a branch of the tree, the number of bound, fully-specified top-level service parameters increases monotonically. But calculating the maximum deliverable utility involves iterating over a set of nodes in the utility space corresponding to those nodes that have the specific values of the specified values and any combination of the *unspecified* parameters. As we move down the search tree, previously bound parameters never change their value but unbound variables become bound.

Because of this, the set of bound parameters is always a superset of those available at any higher level node along this branch, and therefore the number of nodes searched in the utility space must be a subset of the parameters being investigated at any higher level along that branch. Therefore, the maximum utility value can only be lower than that obtained at any higher level node along the same branch of the search tree.

Summarizing, as we go down a branch of the search tree, maximum realizable utility decreases monotonically and cumulative resource cost increases monotonically. Therefore the net-benefit (the difference between these two) monotonically decreases. This is precisely the condition needed for branch-and-bound cutoff.

Note that we're not performing any kind of heuristic for searching the subservices; i.e., we're not trying to sort them by lower resource cost in an effort to perform an earlier pruning of the tree. This is because the only effect a new service can have is on the binding of the top-level service parameters, and that's the only metric that actually impacts the algorithm.

With these changes, we have turned the paradigm for method selection into full decision-theoretic hierarchical task-net planning, yielding a far more robust result. Appendix A.3 contains the modifications necessary to implement this algorithm.

5.4.3 Reliability of Resources

One more wrinkle is introduced when considering the resources – sometimes, they fail. Resources are rarely perfectly reliable, whether through hardware failure (the bulb in a projector has died; a needed computer has been unplugged or crashed) or through software problems (a server has been overloaded; network connectivity problems), or even just simple errors. We can change our calculations to take this into account, by changing the expected benefit to scale the utility according to the probability that all the resources work.

But how do we calculate the probability that a resource will work? One simple estimate is to track failures that are noticeable to the system itself; if a resource call fails (for any reason) or raises an exceptional condition, we mark that information into the permanent storage for the resource information, along with a running total of the number of times the resource has been called. This will give us a rough estimation of the probability that the call will succeed, based on previous history. Although there may be errors that are not visible to the call (for example, a failed projector bulb may not result in a error that is visible within the system), we optimistically choose to believe that such errors are rare and software problems will dominate.

Similar to the resource cost, we also track a resource's "cost of failure", which serves as a penalty for choosing an incorrect resource.

We then change our hierarchical planner so that it uses the probability in the calculation, changing the calculation of utility so that it is multiplied by the accumulated probability that the resources are working, and augmenting resource costs with their failure penalty (scaled by the probability that the resource will fail). Because the probability that all resources work can only decrease as we descend the service tree and add more resources, our branch-and-bound design in the previous section still works. Code that performs this is included in Appendix A.3.

Note that this code optimistically chooses to believe that all failures are independent. We assume, for example, that the probability of failure in sending mail to one server is totally independent from the probability of sending mail to another (when, in actuality, the probability of failure in sending mail may be affected just as much by the local mail handler than the conditions on the remote server). To do this properly would probably require implementing Bayesian networks to get a better estimate of the true joint probability.

Chapter 6

SEMANTIC: A Semantic Network-Based Knowledge Representation

6.1 Knowledge Infrastructure

In order for any IE to operate (or, indeed, for the operation of any computer system that must catalogue or interpret real-world events), it needs a means of organizing knowledge about the world and its inhabitants, and for making inferences based on that information. In our scenario, it is easy to see several categories of information that need to be stored:

- Information on the system's users (Alan, Beth) and simple information about them (email addresses, phone numbers)
- Presence information (Beth is located in a conference room, Diane is elsewhere)
- Location context (Beth's location is currently noted as the room for a currently active meeting)
- Preferences (Diane prefers to get messages via phone)

- Device information (The conference room has speakers and displays)

Of course, any suitably flexible environment must also maintain a great deal more information for other, more specific tasks. For example, the "meeting mode" that the conference room has entered is indicative that the environment is performing a large number of support tasks for a meeting, which may also require tracking and presenting:

- Meeting attendees
- Agenda items
- Recordings of the meeting, through a secretary's minutes, or through any capture devices available (audio, video, whiteboard capture)
- Documents that need to be entered into the meeting record
- Discussion points that get raised
- Action items that get assigned to attendees

...and the list continues. Because many of these items reference other items in the knowledge base (for example, a list of the meeting attendees probably references several other users of the system, like Beth), there is a benefit in being able to store these differing pools of information in a common knowledge repository, rather than dividing them into separate knowledge "fiefdoms" that would need to be linked together by each application referencing the knowledge structure.

Looking at the needs of the IE, we can create a series of requirements for an environment's KR. These include:

- Efficient Update and Retrieval
- Persistence
- Inference Generation
- Transparency

6.1.1 Efficient Update and Retrieval

Since uniting all the different nuggets of information into one store is beneficial to the applications making use of the data, being able to operate on the knowledge store swiftly is an important design goal.

Although this is true for all KR implementations, the necessity is larger with an IE, because the breadth and information is so much larger.

As the number of relationships being stored grows, steps will need to be taken to ensure that adding a meeting agenda item or querying a user's location does not require that the environment be taken offline.

The KR must also be designed with an eye towards the kind of query operations that should be performed. If all operations are along the lines of "tell me what room Beth is in," it is much easier to structure the KR for efficient retrieval than "give me the location of every conference room on the 2nd floor that has a projector". Although such a complex query may need to be performed by the IE, there may be a value in simplifying the request so that the IE has to perform several queries in sequence to filter the result set appropriately. In addition, simplifying the types of queries that can be performed has an additional benefit in that it simplifies the KR interface for the applications (as will be shown in Chapter 6).

6.1.2 Persistence

Because an intelligent environment should operate continually, IE applications need to be designed for much longer lifetimes than typical desktop software applications. To ensure that the IE is responsive to the needs of its users, applications are usually designed to be in a constant event loop, waiting for commands. As such, pieces of IE applications often have a very open-ended design, and don't have a specific "exit" or "shutdown" command. The fragile state of leading-edge hardware and software, however, occasionally means that the operating platform may cause the applications to terminate or freeze without warning.

For this reason, any KR needs to checkpoint its state to a persistent storage device, so that when the IE is revived it can resume functioning where it left off. Although the KR can cache information in memory in order to speed retrieval, any updates need to be written to a persistent medium as swiftly as possible. Updates should also be transactional, so that the representation will never be in a state where a change has been executed only halfway – either the change was committed or it wasn't.

6.1.3 Inference Generation

For maximum utility, the IE needs to be able to easily link together disparate pieces of information in order to assist the user. For example, a user might be interested in learning more about the Lisp programming language, and be using an IE to assist him in navigating through online programming resources. If the IE can recognize that he is trying to gather information on Lisp, it may be able to search through the recorded interests and expertise of the user's coworkers and find one with the appropriate knowledge – or even suggest that the user contact a friend who recently attended a meeting where Lisp was heavily discussed.

Being able to perform this kind of operation requires that the KR used must be able to take into account two disparate pieces of information (a user and a topic area), and generate a series of links that can be presented for user evaluation to determine whether it is useful. Of course, concision is useful here; it is likely more appropriate to find just a few simple links between the two items than a meandering path through the knowledge base that encompasses lots of information.

6.1.4 Transparency

There is also a value in *transparency*. By this, I mean having the format of the knowledge base track as closely as possible the relationships and descriptions that humans use to describe the data – in essence, being able to have as direct a representation for the phrase "George is having a meeting in room 232" as possible. By having the representations match the user's assertions, there is less of a worry of translation errors between the user's view of the world and the IE's knowledge base.

Of course, this is likely impossible to achieve perfectly, since doing so would require a system with perfect recognition of English grammatical structure to make querying useful. However, an attempt can be made to provide a reasonable compromise between strict English sentences and data storage, by translating the English into simpler data forms that express relationships.

6.2 The Semantic Network

To satisfy these goals, we require a KR that can easily encapsulate the many different objects and associations implied by the text, including the people, spaces, location information, devices, meeting modes, preference representations, etc. What is needed is a representation that can store information on objects such as people and spaces, but also can easily track and follow the relationships between them.

One such representation is a "semantic network", which has its roots in Quillian's work on reasoning in computer environments [38]. This work represented knowledge as concept nodes related by directional relationship links, representing the world as a directed graph.

Exploring the framework of knowledge in such a system, therefore, is as easy as moving from a node along one or more links to discover related information. For example, finding Beth's current location might be as simple as starting at the node representing Beth, moving along a "located-in" link to find her current space, and then following the links to determine the device agents currently available in the room. This structure makes it easy for an intelligent system to uncover information about a particular topic, as well as to discover the relationships between two different objects.

The semantic network also satisfies many of the other conditions imposed by the IE. It is easy to encapsulate into human-readable form (either through graph networks or simple text representations); it's simple to add new information in a localized fashion; and making inferences is often merely collecting links and following them.

Much of the work behind semantic networks is being continued on a grander scale by the World Wide Web Consortium as part of their Semantic Web project [5]. Much of that project focuses on addressing the central problem of larger semantic networks – most notably, the problem of unifying large or multiple ontologies.

The problem with multiple ontologies is straightforward to describe, but difficult to overcome. Although it is easy to describe a mapping from a human-centric view of the world (people, places, things) into a set of descriptive names, such a mapping is bound to be highly domain- or location-specific. When different people try to create their own mappings, they either have to shoehorn their own mental model of the mapping into one created by somebody else, or create their own, separate mapping. The Semantic Web project has spawned technologies for translating one ontology into another, so that people can choose either to use someone else's ontology or "roll their own" as they see fit. One such technology is Fensel et al.'s OIL [10] and its more recent offpsring, OWL [40]. Although our current work is not focused on overcoming the ontology problem, we are looking towards this and related projects as extensions to our work.

6.3 Design and Implementation

For the purposes of the Metaglue-based IE, I have developed an implementation of a semantic network featuring the above attributes, which I refer to rather simply as SEMANTIC.

6.3.1 Database

Although SEMANTIC has a modular design which could allow for different backend implementations, the current implementation is designed around a SQL-based backend, built around the MySQL database server[31]. MySQL is an open-source database product, designed more for speed than for full transaction-based 'ACID' (atomicity, consistency, isolation, durability) compliance (although recent versions of MySQL do in fact enable ACID features under limited conditions). The Metaglue semantic network implementation, however, can easily use a different back-end, and has been successfully operated using the ACID-compliant PostgreSQL database [37].

The SEMANTIC database design consists simply of two tables. One of these merely holds the list of active node identifiers in the database. The other is a table of tuples holding the node data, linking node identifiers and keys to the values for each key. Simple information for a person might be stored in a set of tuples as follows:

⟨ 10342, is-a, `semantic.Person` ⟩
⟨ 10342, name, "Bruce Wayne" ⟩
⟨ 10342, email, `bwayne@gotham.gov` ⟩
⟨ 10342, birthdate, 'May 27, 1939' ⟩
⟨ 10342, employee-id, 928340 ⟩

As is implied by the above, the data for the value can vary widely in type, from character strings to dates and integers. Since SQL databases require that the columns be strongly-typed, all these values are turned into character string values for storage.

For all objects, there is a special key, is-a. This defines the type of the object. This is usually the name of a Java type for the object, thus providing a simple ontology for the system to use and making it easier to write Java-based Metaglue agents that can utilize the system. As a side benefit, the Java type is used to guide the interpretation of the value fields.

The database is indexed based on the unique identifier code and the tuple key (in the above example, the first and second columns of the tuples). The key values are by default not indexed, although adding in an indexed column by hand has been found to have benefits.

6.3.2 Java-Based Interfaces

All SEMANTIC types inherit from a single superclass, UniqueID, which provides for a Java object with a unique identifier code. Users creating a new node type for the semantic network can do so simply by creating a subclass of UniqueID, and can then install instances into storage simply by calling the updateObject() method. That method will ensure that the database has all the proper values for the node keys.

SEMANTIC uses the Java reflection API in order to determine the set of keys to install for a given node. Any Java field that is not marked as special (i.e., that doesn't have the "transient", "static", or "final" modifiers set) are automatically stored as tuple sets into the SEMANTIC storage. Fields that contain other UniqueID objects are automatically converted into node references, and the referenced object nodes are stored or updated at the same time. Other useful node types, such as dates, numeric values, or booleans, are converted to and from strings as appropriate. Special handling is also provided for standard Java arrays, as well as members of the Java Collection framework, to also automatically store those fields as tuple sets linked to the node identifier.

Similarly, the SEMANTIC links are all subtypes of a Link subclass of UniqueID. Link provides for the simple linking together of two different nodes using "from" and "to" fields, which, just as with all UniqueID subclasses, get converted into tuples as described above. Because Links are also Java objects, they can contain other information that can be used to describe the parameters of the link – for example, allowing agenda items to be linked to a specific meeting together with a status code describing the importance of the item in that gathering (since often the same item may appear in different meetings, but have differing importance depending on the meeting focus).

All UniqueID objects can be queried directly by object ID, or by finding a set of objects that are similar to a provided prototype. For the prototype matching, a programmer can create an instance of a node class, set values for fields that he

```
Person test = new Person();
test.setEmailAddress("bwayne@gotham.gov");
List l = test.findSimilarObjects();
```

Figure 6-1: Java code to get people who have a given email address

```
ConditionCollection cc = new ConditionCollection();
cc.addRange("startTime",
            new TimeStamp(early.getTime()),
            new TimeStamp(late.getTime()));
List lst = UniqueID.findObjects(cc);
```

Figure 6-2: Java code to find objects that have a "startTime" field between two given times.

wants to use as match parameters, and call the findSimilarObjects() method to get a list of objects that match the parameters. For example, the code in Figure 6-1 will locate the set of objects in the network that are of type (or subtype) Person, and which have an email address field set to "bwayne@gotham.gov".

More complex queries can be constructed using the findObjects() method and its associated classes, ConditionCollection and TuplePattern. ConditionCollection holds a simple collection of clauses that allow for more intricate query terms, such as finding all objects with keys that are greater than a given sentinel value, or specifying a discrete set of candidates that a value can match, or a range that a value can fall into (see Figure 6-2). The elements of the condition can be specify either exclusive (OR) or inclusive (AND) matches in the result to allow for more complex node queries.

TuplePattern is also based on the notion of providing a set of clauses for complex node setups, but acts on the base tuple sets of the database, in much the same way as the query interfaces described in the next section.

6.3.3 Query Interfaces

Although the Java interfaces allow for querying of nodes using the findSimi-

```
List out = new ArrayList();
Group grp = new Group();
grp.setName("MyGroup");
List l1 = grp.findSimilarObjects();
if (l1.size() == 1) {
  Group g = (Group)l.get(0);
  MemberOf m = new MemberOf();
  m.setToNode(g);
  List l2 = m.findSimilarObjects();
  Iterator iter = l2.iterator();
  while (iter.hasNext())
    out.add( ((MemberOf)iter.next()).getFromNode() );
  return out;
} else {
  throw new Exception("found more than one match");
}
```

Figure 6-3: Java code to get people who are members of a given group.

```
(ask '((?x is-a semantic.Group)
       (?x name MyGroup)
       (?y is-a semantic.MemberOf)
       (?y fromNode ?x)
       (?y toNode ?z)))
```

Figure 6-4: Pattern-matching code to get people who are members of a given group.

larObjects() and **findObjects()** methods, there are drawbacks with that approach. More complex queries inevitably require the use of several different objects all linked together. For example, to find the set of people who are members of a given group, you would first need to query for a node representing the group, then query for membership links that point to the group node, and then return all the source nodes in a list (see Figure 6-3). This code can be fairly complex, and dealing with odd conditions makes it even more so.

To alleviate this, a rule-based pattern matcher was written in Scheme using the Java-based Kawa Scheme engine ([6]) and implemented to make queries like this simpler to write and easier to understand (Figure 6-4). Because the pattern matcher

```
(define-rule links
  ((?l is-a ?type)
   (?l fromNode ?f)
   (?l toNode ?t))
  =>
  (?f (link ?type) ?t))
```

Figure 6-5: The "links" rule for the query engine.

```
(ask '((?x is-a semantic.Group)
       (?x name MyGroup)
       (?x (link semantic.MemberOf) ?z)))
```

Figure 6-6: Using the links rule to get people who are members of a given group. Note that this will perform the same search as in Figure 6-4

is rule-based, we can also create a links rule (Figure 6-5) which can make queries involving links a bit easier to write (Figure 6-6). In this instance, the backward-chaining rule engine will search backward through the rules to resolve the (?x (link semantic.MemberOf) ?z) into the correct result.

In response to concerns from the Ki/o kiosk effort, a new Java query interface that provides some of the utility of the query code has also been provided. Although this interface does not employ the rules engine of its Scheme-based counterpart, it allows a piece of Java code to perform very efficient searches in much the same way as the Scheme engine, but without the overhead of running the Kawa subtask. An example can be found in Figure 6-7.

6.3.4 Triggers

Both the Java and the query engine have support for triggers; that is, allowing for code to be called when a new piece of information is added into SEMANTIC. On the Java side, there is the TupleListener interface, an method definition which can be implemented by any agent in the system to enable a callback for new or updated tuples in the SEMANTIC storage. The Java code then can perform further tests on

```
TuplePattern tp = new TuplePattern("z");
Object x = TuplePattern.mark("x");
Object y = TuplePattern.mark("y");
tp.addPattern(x, "is-a", "semantic.Group");
tp.addPattern(x, "name", "MyGroup");
tp.addPattern(y, "is-a", "semantic.MemberOf");
tp.addPattern(y, "fromNode", "semantic.Group");
tp.addPattern(y, "toNode", TuplePattern.mark("z"));
List lst = UniqueID.findObjects(tp);
```

Figure 6-7: Java-based pattern-matching code to get people who are members of a given group. This will perform the same search as that in Figure 6-4.

```
(define-trigger
  (send-email-for-discourse (?link fromNode ?d))
  ((?link toNode ?t)
   (?d is-a applications.meeting.DiscourseItem)
   (?t is-a semantic.Topic)
   (?e email-interest ?t))
  =>
  (printout (quote ?e) ": " (invoke ?d 'getSubject)
            " is part of topic area "
            (invoke ?t 'getName) ))
```

Figure 6-8: Trigger definition for reporting on new meeting information that matches a given topic.

the matched tuples to determine whether it can make use of the new information.

Similarly, the query engine allows for trigger bodies to be created based on new or updated information, and further allows the rule engine to perform better fine-tuning of the results, automatically creating a rule which will fire when a new tuple is created, but only when the tuple matches information already in the system (Figure 6-8).

6.3.5 Requirements Utility

Turning back to our list of requirements for the knowledge representation, how well does this representation address them?

Efficient Update and Retrieval As long as the basic operations of creating nodes and links are reasonably efficient, the efficiency of updates is similarly so. Since each node and each relationship is represented just once, the update efficiency is related closely to the more primitive creation operations.

In SEMANTIC's implementation, a well-indexed SQL backend can perform highly efficient updates and additions to the data store, resulting in updating times proportional to the number of tuples used in the node or link.

Retrieval operation efficiency is closely related to the indexing used in the SQL database. Finding information when the UniqueID's unique identifier is already known is relatively fast; if a node needs to be searched for based on other information (for example, looking for the person whose email address is `"bwayne@gotham.gov"`), is highly dependent on the indexing of the tuples. Providing a full-text index on the key values can speed this up substantially.

Persistence By constantly storing information into a persistent database, this requirement is easily satisfied. The SQL backend is constantly checkpointing the state into a filesystem. The SQL framework also assists with making sure that updates are performed in an atomic fashion, and that objects are not stored in a half-completed form in the database.

Inference Generation The TuplePattern design is used to link disparate objects together, and allows for easy generation of links between two items and efficient querying of those links.

Transparency Although this is difficult to quantify, the storage format allows users or programs that wish to pull out the right information to write small queries that will easily get the necessary match. In addition, users can benefit from simple graphical visualization tools (see Section 6.5) which show the network as an easy-to-read graph.

6.4 Usage

The SEMANTIC implementation is currently used as the basis of several projects, including the Computer Science and Artificial Intelligence Laboratory (CSAIL) OK-net project[17], which is deploying kiosks throughout the CSAIL building to enhance information dissemination and collaboration. That project uses SEMANTIC to store hundreds of notices and news items, as well as information about people and places in the lab, and then links them all together according to people's interests and the kiosk's location.

While developing this system, we have identified several areas in which using a semantic network-based representation is an appropriate and valuable piece of infrastructure for intelligent environments such as this one, most notably in the areas of user information, meeting management, and location infrastructure.

6.4.1 User Knowledge

One of the key pieces of knowledge for any intelligent space is that of the users and the individual spaces they work with. At a simplistic level, this can simply be a set of objects comprising spaces, how they are encapsulated within each other, and the user's current location. This gives access to straightforward queries like "where is Steve located" and allows for simple resource management dependent on the task and space involved [11].

However, in order to make a system that truly acts as an "intelligent assistant", you need to include far more information about the people and their relationships. Such systems need to be able to respond to queries and requests such as:

- "Send this information to everyone in the group."
- "Who is Joe's supervisor?"
- "Do I know the person who is responsible for this group's activities?"

For this, we augment SEMANTIC with information on groups of people, and the

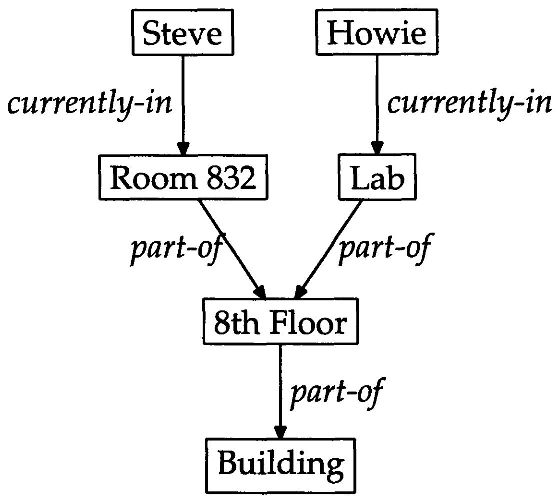


Figure 6-9: Simple Layout for Users and Spaces

relationships between people, including notations about responsibilities and hierarchies (see Figure 6-10). In addition, we are providing information on interests and expertise to enhance "intelligent assistant" roles for the IE. Such information will enable our agent systems to respond to more complex requests for information, and provide introductions to enhance communications between users:

- "Send this information to group members interested in HCI."
- "Do I know anyone who is an expert in writing LISP code?"
- "Who do I know who can introduce me to a LISP expert?"

6.4.2 Meeting Management

Some applications now use SEMANTIC to capture meetings as they occur, linking together the main meeting topics along with their contributors and attendees. Typical information that gets captured during a meeting includes agenda topics, action items, supporting and dissenting arguments, and documents such as presentations or web references. People are linked in as meeting attendees, document authors,

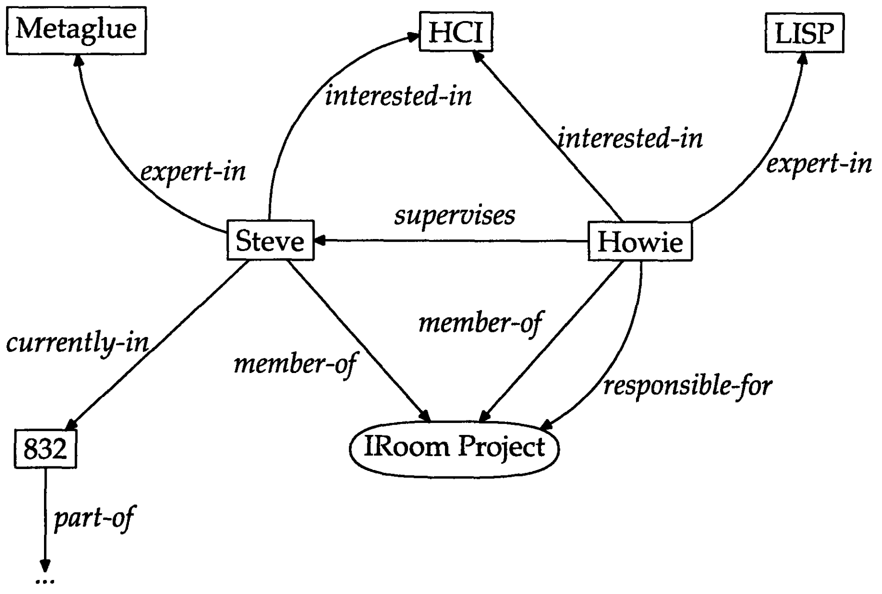


Figure 6-10: Extended layout, adding information on interests, expertise, hierarchies, and groups

and as issue-raisers. When meetings take place in instrumented environments, they can be linked to a video or audio capture of the meeting in progress.

Using the philosophy that meetings are not the primary piece of information, but merely a framework for examining and disseminating information, the discussion topics within the meeting management software can be linked together. This makes it possible to review proposals as they travel through a long-term set of meetings, and to ask the system questions regarding previous meetings discussing the current topic.

6.4.3 OK-net Project

The OK-net[17] project also makes liberal use of SEMANTIC to store information about people, places, and relationships between them, in order to support collaborative event filtering and to provide directory and guidance operations for the CSAIL organization from several. The project also stores and organizes information on thousands of events occurring on the MIT campus.

6.5 Visualizing Semantic Networks

One of the nice aspects of the semantic network is that it provides for an easy translation from a network fragment into plain English representation. For example, the fragment found in Figure 6-9 clearly represents a handful of English sentences:

- Steve is currently in Room 832.
- Room 832 is part of the 8th floor.
- The 8th floor is part of the Building.
- Howie is currently in the Lab.
- The Lab is part of the 8th floor.

With simple knowledge of what a link between two objects means, and the name of any given object, any network fragment can easily be converted between natural language and network representations. Indeed, systems like Katz's START system[15] can already perform a decent job of translating from English into a set of simple tuples like those that we show in Section 6.3.1.

Interfaces to examine the network in a visual manner are also highly useful. As such, applications which allow a user to explore the semantic network as interconnected graph nodes can be useful. I've created one such application, based on the TouchGraph [41] graph manipulation tool. The tool can be set to only display a bounded selection of nodes, centered on one element in the network – that is, displaying only those nodes that are a maximum of, say, three links away from the focus (see Figure 6-11). Nodes that are at the periphery of this "locality" are notated with the number of links that are connected to them.

More elaborate graphs can also be printed offline by generating large maps of data and turning them into images, using AT&T's GraphViz software layout tool [9]. For an example of this, see Figures 6-13 and 6-14 later in this chapter.

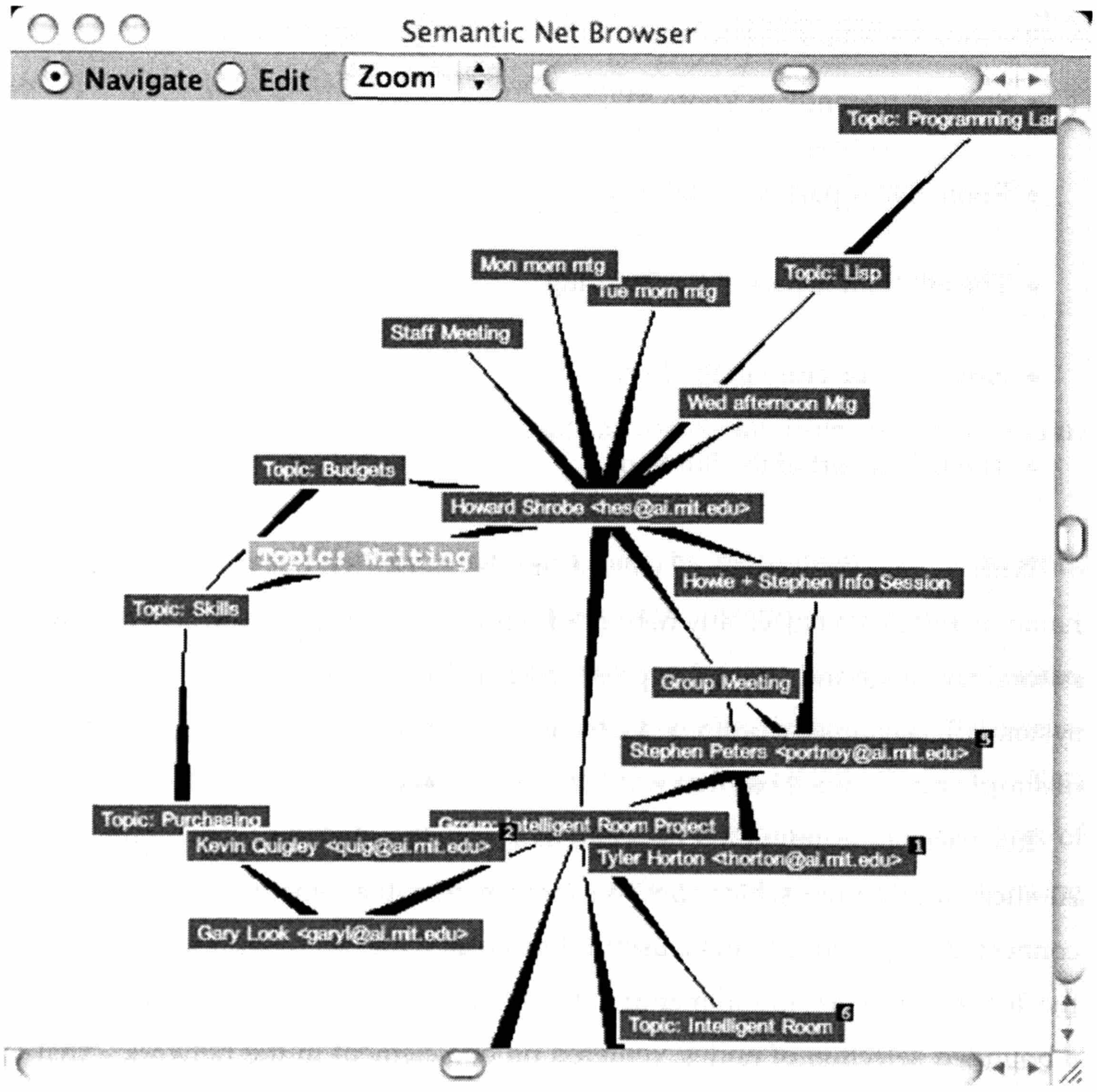


Figure 6-11: The Semantic Network browser interface

6.6 The Meeting Management Application

It may be useful to examine one of these applications in depth to get a real sense of how the network gets created and what it can provide. For this reason, we will go into some detail here about how the MeetingManager application utilizes the tools of the semantic network and the Intelligent Room.

The Intelligent Room project uses the Java-based Metaglue agent infrastructure [8] to build agents that can communicate with each other. These agents are identified by their function, so that, for example, the agent that activates projection screens within a room is called the ProjectionScreen agent. For the most part, agents communicate with each other through direct, one-to-one remote method calls, although broadcasting facilities are available.

The current version of the MeetingManager application builds upon initial work by Oh et al. [33], who created an application to demonstrate the use of an intelligent room in a collaborative meeting context. That application maintained information about the agenda, major issues raised, and any commitments agreed to during the meeting time. It could also record information to link in these events with a QuickTime recording of the session. Although it performed well as a demonstration, it was hampered by a lack of robustness in the data model, with an inability to capture anything deeper than the broadest points of a discussion, and no capacity for reviewing and augmenting the discussion off-line.

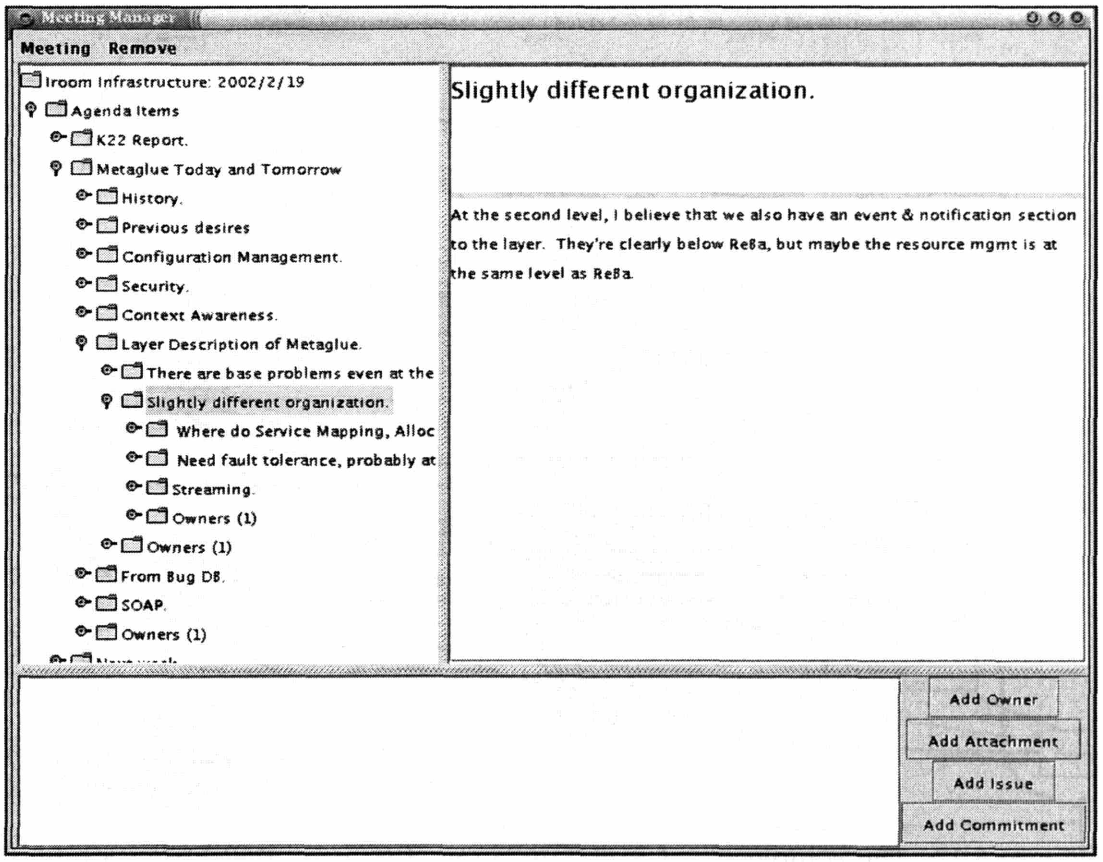
This evolution of the MeetingManager application encompasses several agents that act in concert. The first of these, MeetingModel, serves merely as an interface to the semantic network and abstracts out some of the network lookup tasks into simpler methods. It also has the ability to broadcast any changes made to the meeting structures to other agents that request them. We suspect that this will likely be a design feature of many of the Intelligent Room's semantic network applications, since having one agent which can coordinate and monitor the KR for appropriate changes prevents duplication of code and provides for better data abstraction.

Other agents are available which provide different interfaces to the meeting manager data. One, the Gui agent (see Figure 6-12), builds a tree-structure view of the meeting information and allows easy creation and editing of topics, issues, and discussion points. This editor serves fairly well as a primary interface for a meeting recorder as he or she takes notes. Currently, this is the primary conduit for meeting information to get into the semantic network, although we expect that in the future our IE implementations will add in some of this information itself (such as automatically recognizing meeting attendees and using knowledge about seating arrangements to determine who is raising the discussion points), and to provide more robust interaction with the room (such as using more voice commands to create nodes).

Other viewing and editing agents provide different views of the data – through web interfaces, graph networks, or specialized meeting views for presenting the agenda items or commitments. All of these are updated as information gets added to the semantic network. Meeting attendees can decide to bring up their own person view of the meeting state, so that they can augment and add the meeting information with more information than the meeting recorder was able to provide.

The network itself stores a variety of node and link types for the meeting, including meetings, multimedia recordings, documents, people, issues, commitments, and other discourse items. Most of these types actually have very little information attached to each of them, as the semantic network relies more on the links between nodes to define the relationships. For example,

- a person can act as an *author* of a document, an *attendee* of a meeting, a *facilitator* for a meeting, or an *owner* (sometimes considered the "raiser") of a discourse item;
- discussion points can be raised as a *supporting* argument, a *dissenting* argument, or even as an *implication* to a preceding discussion;
- discourse items can be marked as *agenda* nodes so that they can be used to organize meeting flow.



Iroom Infrastructure: 2002/2/19
- Agenda Items
  - K22 Report.
  - Metaglue Today and Tomorrow
    - History.
    - Previous desires
    - Configuration Management.
    - Security.
    - Context Awareness.
    - Layer Description of Metaglue.
      - There are base problems even at the
      - Slightly different organization.
        - Where do Service Mapping, Alloc
        - Need fault tolerance, probably at
        - Streaming.
        - Owners (1)
      - Owners (1)
    - From Bug DB.
    - SOAP.
    - Owners (1)

Slightly different organization.

At the second level, I believe that we also have an event & notification section to the layer. They're clearly below ReBa, but maybe the resource mgmt is at the same level as ReBa.

Figure 6-12: The MeetingManager's Gui.

The endpoints for the links are not limited to the storage nodes; links can also be created to other links. This allows more complex interactions in the data model, so that a person can augment a supporting argument with a node describing his own agreement. We use this ability to augment links extensively to register that a link was created during a meeting, so that we can later request information on any discussion points raised at a certain time, and link those points to the meeting video.

With many discussion points being raised during a meeting, these networks can grow to be fairly complex, as seen in Figures 6-13 and 6-14.

When the meeting is over, the network makes it possible to delve into the structure and answer many questions about the meeting, among them:

- "What members of the AIRE group did not attend the meeting?"

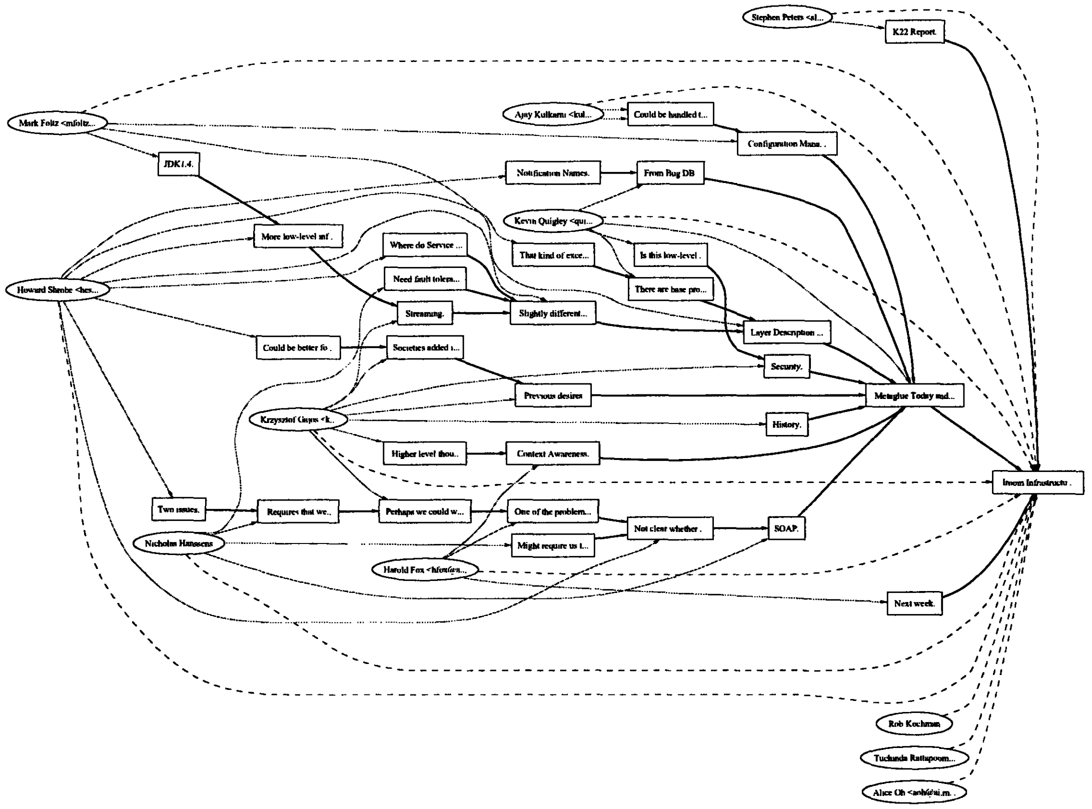


Figure 6-13: The complexity of a discourse structure captured during a typical "brainstorming" meeting. People are represented by ovals, discourse items by rectangles. Different relationships are represented in different colors. A closeup of this is in Figure 6-14.

- "What open commitments are assigned to me?"
- "What issues were raised in opposition to this discussion point?"
- "What points did Krzysztof raise?"
- "Show me the video for this discussion point."

The last example works by coordinating the timestamp for the discussion points with that of the meeting video, so that it can skip forward to the appropriate position during the video playback. By linking together multiple meetings, it is also possible to make these queries about the meeting history, and retrieve the archived footage from previous discussions, allowing for a quick recap of the important events.

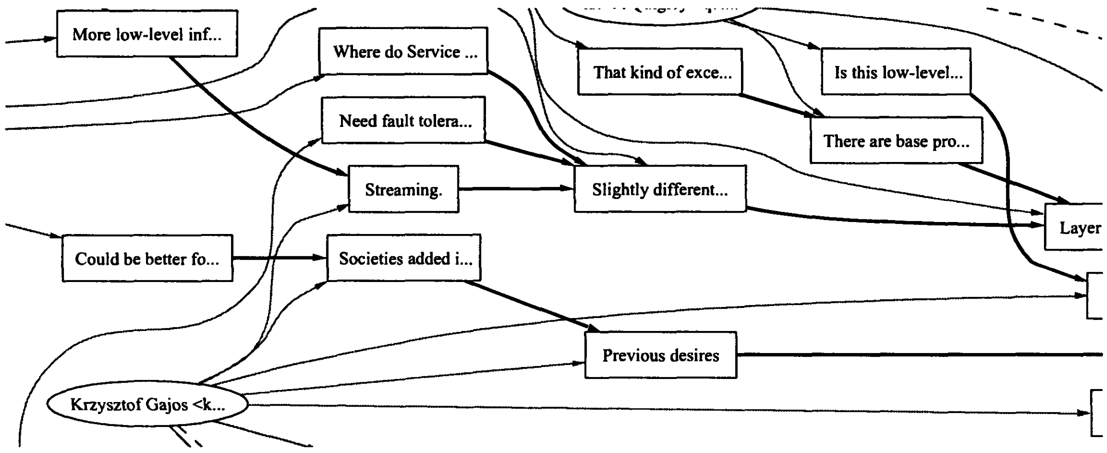


Figure 6-14: Closeup of Figure 6-13. The links from users to discourse items specify the "owner" of each discussion point, whereas links between discourse items are discourse links.

Chapter 7

Evaluation

Recall the design criteria referenced in Chapter 1.1:

- **Respect people's right to control their own resources**, by ensuring that resources are controlled only by the resource's owner(s), and can enable restrictions on the resources as they see fit.
- **Respect people's privacy**, by limiting access to private information (such as a person's location), and only allowing such information to be retrieved from the outside by querying data sources controlled by the person in question.
- **Adapt to changing conditions**, so that the environment can adjust its actions dependent on what situation a user finds himself in, and
- **Be sensitive to individual preferences**, such that different people can guide the system in different ways based on their own personal needs.

How well does the system we have architected handle these constraints? Let us describe fully how the entire system handles the donut scenario:

- There are several societies in the system. In particular, there are societies for Max, the lounge, and Max's workgroup (which we shall call "AIRE").
- The lounge's semantic network contains a list of the groups that are associated with it, including the "AIRE" workgroup.

- Ellie uses the lounge's kiosk to send an announcement message to the locally associated workgroups. This announcement message has a few parameters because of the nature of a "free food" broadcast – it is important that the message be delivered quickly (because can disappear fast), but it is not an urgent message (so no guarantees of delivery need to be performed). As such, the message is tagged with parameters of "high timeliness" and "low urgency."

- The agent in the kiosk society which receives the request decides to perform a multicast to all associated workgroups, passing on the message and the same delivery parameters. As such, it looks up the list of groups that are associated, and requests an announcement service for each one. Each request is handled in the same way, but we shall here examine only the request being sent to the AIRE society.

- Because the request is for a foreign service, the request is forwarded to the ambassador for the lounge society. The lounge's ambassador notes that the request is being sent to the AIRE society, so it consults the Hyperglue Entity Directory, and requests a handle to the ambassador for AIRE.

- Once the lounge's ambassador receives a handle for AIRE, it passes on the request for an announcement service to it.

- The AIRE society is a group society (Section 4.6), and as such often passes on messages to members of the group. However, it does have two means of handling announcements to the group:

  - A local announcer, which will try to locate and use area-wide message boards or scrolling displays to pass on announcements and news items. Because people only really look at it when they're passing by, this service has parameters of low timeliness and low urgency.

  - A multicast announcer, which will re-send the announcement individually to each member of the group. This service is considered to have a high timeliness factor, since it will tend to be received faster.

- The ambassador forwards the service request on to the AIRE service mapper. Because the incoming announcement message has a high timeliness factor, the service mapping engine in the AIRE society evaluates the multicast announcer as having a better utility, and thus returns it.
- The ambassador for AIRE will then pass a proxy to the multicast announcer service back to the agent in the lounge society. The agent in the lounge society can then use this proxy to call the AIRE announcer with the message attached.
- The multicast announcer in AIRE receives the message request, and performs its function – looking up the list of users in AIRE, and forwarding the message on to each user's preferred announcement service. As such, it looks up the list of users that are members of the group, and requests an announcement service for each one. Each request is handled in the same way, but we shall here examine only the request being sent to Max.
- Because this request is for a foreign service, the request is forwarded to the ambassador for AIRE. Because the AIRE society ambassador notes that the request is being sent to Max, so it consults the Hyperglue Entity Directory, and requests a handle to the ambassador for Max.
- Once the AIRE ambassador receives a handle for Max's ambassador, it passes on the request for an announcement service to it.
- Max's society contains several means of sending out announcements:
  - An SMS-based service, which will use an email-sending resource to deliver a text message to Max's cell phone. Since Max always has his phone with him, this service has a high timeliness and high urgency.
  - An email service, which will use the same email-sending resource to deliver a text message to Max's email inbox. Max doesn't check his email

  very often, so he gives this service a low timeliness parameter. It does, however, have high urgency, because he tends not to lose email very often.

  - A display service, which will coordinate with the local environment at Max's location and try to display a message there. Max usually notices these messages, but doesn't like important information to be sent there. Therefore, he has marked this service as providing high timeliness, but low urgency.

- Max's current context contains information that he is currently not at his desk. His contextualized preferences are stated such that if he is currently at his desk, he prefers urgency over timeliness, but when he is away, he prefers timeliness over urgency.

- Max's preferences in this situation are turned into a utility function, and each of the services are evaluated. The SMS-based service is evaluated as returning the highest net benefit. As such, a proxy for that service is returned to the AIRE society.

- The AIRE society's announcment service then uses the proxy to pass the message on to Max's SMS-service.

- The SMS service requests the email delivery resource, uses Max's local semantic network to get the email account that it should send the message to, and then uses this information and the resource to pass the message on to Max's phone via the email resource.

- Everything after this is up to Max. He gets the message on his phone, reads it, and then goes to grab food.

How well does this system handle the different design criteria?

7.1 Respect Privacy

This system respects privacy by carefully compartmenting the information. All information for a society (Max's location, his telephone access number, his preferences, the list of groups associated with the lounge, the list of members in AIRE), is stored in local semantic network storage, and thus is not leaked out to the wider world. Although external societies could request this information, deciding whether or not to respond is done by agents under the control of the information holder.

7.2 Ownership Rights

Decisions to use resources are made by the resource's owner – for example, Max's email delivery resource is only accessible to the local society, so it cannot be grabbed or misused by external societies without permission. Max (or agents operating on his behalf) also gets to decide precisely how messages get delivered – the sending agent can try to make requests, but the final decision is up to Max's society of agents.

7.3 Changing Conditions

The contextual parameters of the situation guide which set of preferences are used in the service mapping engine. When Max is in a different location, the contextual situation changes as well, and thus a different set of preferences are called into play. With different preferences comes a different set of rules, and potentially a different service.

7.4 Individualized Preferences

Unsurprisingly, the system handles this well, since it exists directly as one of the four pillars of infrastructure. Preferences are stored locally, and used to guide the selection of the most appropriate service for any task.

7.5 Summary

In sum, the system as designed does in fact satisfy the design goals.

Chapter 8

Future Work

As no research project is ever completed, most of the different components of this work present opportunities for further examination. In the following sections, I will briefly outline several possible areas for improvement or analysis.

8.1 Access Control

The design of Hyperglue, focusing as it does on individual control over resources, lends itself well to a design that provides for access control that is easily defined by the individual as well. By segmenting resources along a broad notion of "ownership," the society of agents that provide access to the resources can exert an influence over who can use them.

For example, a conference room's staff may wish the room to enforce that projectors are only controllable by the person currently signed up to use the room. A simple access control to satisfy this condition would be for the Ambassador to only release plans for devices to societies that represent said person.

However, more complex schemes can be envisioned, often related to the current contextual situation. When no one is signed up to use the conference room, do the projectors default to public access, or are they locked down to protect the bulbs from wear?

Also, are there others who can override the usage requirements? Although in this thesis we suggest that the "owner" of a resource is usually limited to the person or space that has the most direct control over it, in reality the situation is more fluid. A computer in an academic group is in some sense "owned" by the person who uses it on a daily basis, but in other senses might be considered owned by the group's director, the lab that the computer is part of, the academic department, or whatever external organization provided the funding to purchase the equipment. Not all of these will want to have control over the machine's day-to-day use, but there are certainly times when the group's director will want to take over the machine for a processing task – or even ask that it be made available for use by other people in the group. Defining what people are "allied" with the resource is highly dependent on the task being performed and the requestor, and there need to be more flexible means of specifying the rules and enabling this functionality.

One other difficulty with Hyperglue is the need to limit access based on the plan's requirements. In Java, once you have a handle to a resource through a remote object, you can call *any* method on it, not just those related to the current plan. A better access control system would be able to leverage Metaglue's proxy handlers to limit access on the level of individual methods, thus confining external socities to pre-approved interactions with the local resources.

Design work on both role-based access and method-level control in Hyperglue has been started by Buddhika Kottahachchi [18] as part of his master's thesis. Designing more context-based methods such as those described above would be future work, potentially based on the Kottahachchi design.

8.2 Plan Monitoring

The plan model currently adopted for Hyperglue provides a series of steps that will accomplish the intended goal. However, it eschews an important piece – monitoring of the plan's execution to record the steps being taken and determine that it is proceeding correctly, and to signal the system that the plan has been followed to completion. In some higher-order plans, it may even be necessary to request that the user perform a step of the plan (such as switching on a device that is not available through computer control), and wait for the user to respond.

In addition, plan monitoring should determine whether the plan has been executed satisfactorily, perhaps by examining the device state to see if the parameters are within expected boundary conditions. If this is not the case, then control should be passed to a diagnosis and recovery unit to remedy the situation, and potentially to develop a new plan for correcting or completing the task.

Some of this information is handled through the Planlet[16] system, although a good direction for future work would include the implementation of a model-based diganosis and recovery system which can take advantage of the plan representations to determine a proper way of recovering from errors.

8.3 Reactive Event Processing

The design presented in this thesis is based solely around a request architecture – a society makes a service request, which will eventually generate a specific set of steps to go through in fulfilling that request. This setup works well for human-controlled actions, where a person makes an affirmative request and receives a response in turn. However, it works less well for more implicit actions – ones that are initiated by the environment itself, usually through some kind of external stimuli such as someone entering a room. The problem with any such implicit action is that an automated action that is correct under some conditions is often wrong under others. For example, turning on soft music when the lights are dimmed might be quite pleasant when someone is trying to relax, but is rather a poor outcome when the lights have been dimmed because a movie is being viewed.

A few years ago, Ajay Kulkarni[20] described a design for a context-dependent IE, which would model the environment's current state according to its *activity context* – that is, the ongoing task – and describing a set of behaviors for the environment based on that context, which determine how the IE will react to external stimuli. For example, the system might determine that a person entering an empty room should cause the lights to turn on, but a person entering a room where a presentation is being given should cause no such change in light level. Behaviors were modules that could be inherited (so that a "Presentation" behavior might inherit from a more generic "Meeting" behavior).

One possible extension to Hyperglue is to provide support and integration for this kind of event-based reactive processing, and integrate it so that it includes support using the existing contextual information in the semantic network.

8.4 Language Independence

One of the problems with Hyperglue now is that it is highly dependent on a Java-centric view of the world; all agents are written in Java, the communication model is based on the Java-specific RMI framework, and we use RMI to perform method calls as well. Because Hyperglue effectively insulates the different societies from each other, however, it should be possible to use a separate, more language-agnostic communication structure for inter-societal communication. This would have several benefits, most notably plug-and-play interactions with other agent and software systems. This would also allow a closer interaction with applications that provide interaction APIs in other languages (for example, many off-the-shelf applications allow control through plug-ins that can be written in C, but don't provide a Java implementation).

To implement this, it is necessary to strictly define the kinds of interaction that will be allowed (essentially, societal registration in the HED, and the Ambassador interface), and to modify the Ambassadors to use an agnostic communication protocol for communication, such as the World Wide Web Consortium's SOAP. Strictly defining the layout of plan objects will also be necessary. Services that provide resources will of course need to be modified to use the agnostic protocol, but the Ambassador could be tasked to set up proxies that handle the translation between the external protocol and whatever communication is used within the society.

A further extension could be completely separating the external communication from any specific protocol; instead, when an external society opens a connection to the Ambassador, it also provides a set of protocols that it can use for communication, and the two societies can then negotiate to determine the best protocol for communication.

8.5 O2S Integration

Similar to creating a language-independent basis for Hyperglue is helping to adapt some of Hyperglue's concepts to other systems. One such system is O2S[32], another project attempting to create infrastructures for distributed systems, focusing on goal-oriented planning and stream-oriented data processing. As such, it has its own design for service mapping architectures, which has proven to be a utilitarian system in tests.

Like the service mapping component of Hyperglue, the goal system of O2S divorces the actual execution of the code from the choice of the best means to service a request. It also builds up a full goal tree and explores it, similar to the fully hierarchical version of the task planner seen in our Section 5.4.2.

One piece of the picture that O2S does not address, however, is how to deal with the larger world. Specifically, it does not have a service discovery system meant to scale to the larger world. In this respect, the society-based model behind Hyperglue can be used to augment the O2S design and allow it to scale to a larger world. Work along these lines may commence soon in a project associated with the T-Party project at MIT's CSAIL.

8.6 Miscellaneous Improvements to Hyperglue

Since the semantic network has become a fairly well-used component within the AIRE group, there are several ongoing projects to try to extend or improve that work. Graduate students Buddhika Kottachchi and Andy Perelson recently engaged in a research project to speed up the network's operation, concentrating mostly on providing a separate table to handle the class identifiers and using it to hasten the queries. For the work on OK-net[17], Max van Kleek developed an ontology generation language called OntoGen that makes it easier to develop a set of classes for the network based on simple descriptions of types and property values. Finding efficient methods for distributing the network ala distributed database systems are also a useful research direction.

The service mapping component leaves one important facet untouched – *time*. Specifically, in long-lived IEs, agents seem to develop a tendency to request a resource and then not releasing it when it's not needed, either because they've developed an error and aren't properly handling the condition, or simply through programmer laziness. In addition, devices can malfunction and cause a controlling agent to enter a long-lived wait state.

Because of these issues, it's necessary for the resource management and service mapping components to mandate expiration times, after which the resource being used will be assumed to be available for reallocation. Such a design also enables resources to be briefly yanked away from an existing process for a short-term loan to another application. For example, while a movie is playing, an agent may wish to briefly control the speakers of the room to inform the occupants of an emergency. The room's resource manager could briefly wrest control of the audio from the movie-playing agents, and then return it when the announcement is completed, without informing the movie agents that they may wish to send the audio to another channel.

The current design for plans uses a simple structure that merely encodes service requests and methods to call on the returned services. A more flexible structure would use Planlet[16], a middleware layer that represents plans and progress through them during execution. Planlet also allows plans to be defined hierarchically and for plans to be modified as they are being carried out. These two features allow plans to be reused by other plans and also allow for steps to be "late-bound" to a plan, if it is beneficial to defer the choice of a specific plan of action until execution time.

Hyperglue's current design for the HED is fairly simplistic, and we need to evaluate designs that scale better to very large national or global hierarchies of societies. Hyperglue also requires more testing in real-world settings, since most of the work on it has been in small test environments. Part of this is the migration of existing applications to the Hyperglue design; although this is actually a simple operation (usually just ensuring that an agent will start in a spatial society rather than a user society), it sometimes requires an analysis of the existing application's design to yield the best fit.

Chapter 9

Related Work

Although other systems have been built with the intention of exploring the interaction of humans and intelligent environments, few have attempted to do so with the intent of resolving the design criteria described in this thesis. Here we present a brief survey of related work in the field of intelligent environments.

9.1 Interactive Workspaces

The Interactive Workspaces group at Stanford is well-known for developing iCrafter [36] and related systems for handling interactions in a single intelligent space. Their infrastructure, however, is very much geared towards a single environment – usually a shared workspace outfitted with large displays and other display resources. Although they perform interactions with more user-centered items such as laptops and PDAs, thjeir model is focused on the room, and not on the users controlling the devices. As such, notions of a user's rights or privacy are not explicitly handled.

9.2 OAA

The Open Agent Architecture [25], one of the more venerable agent systems available, is designed specifically for agent-based interactions. Within OAA, all communication is funneled through a central agent called the *facilitator*, which acts as a coordinator for cooperative problem-solving. Agents register their capabilities with this central agent, and make requests in the form of "tasks," which the facilitator can break down into subtasks and farm them out to client agents that can perform the required actions. OAA itself, however, does not have special facilities for performing resource reservation or for arbitrating service requests.

9.3 Smart Classroom and SmartPlatform

The SmartPlatform infrastructure is currently being used by the Smart Classroom project [45, 46]. This project originally was based on OAA, but recently moved to a new approach, featuring a hybrid communication scheme: short messages are sent through a central broker (the "DS", or directory service) using a publish/subscribe mechanism, but peer-to-peer connections can be created when higher bandwidth requirements are necessary. SmartPlatform itself does not contain facilities for handling or processing resource conflicts, and is beginning to work on gateways to other agent systems (such as Metaglue, described below), to help address some scalability issues.

The infrastructure is used as the basis of the Smart Classroom project, enabling tools for coordination of both remote and local students in relation to a single teacher. However, the system is not designed to handle more than one centrally located classroom; all resources are either local or communicate solely with the central location.

9.4 EasyLiving

The EasyLiving project [7] at Microsoft used a distributed programming environment with a handful of agents acting to manage a single environment. Although they made strides in exploring and modelling user interactions within a smart room, and building reactive behaviors into the space, there does not seem to be an attempt made to move beyond a single-environment model in their system.

9.5 Other Multi-Agent Systems in Intelligent Environments

There are other multi-agent systems that have been employed for intelligent environments, but these systems are often so specialized that they don't have specific notions of users at all.

The IHome project at UMass [23, 22] was a simulated agent architectures for intelligent environmental control, emphasizing resource coordination. High-level agents were capable of recognizing resource conflicts, and prioritizing allocations according to the user's preferences. IHome's agents, however, are designed to work with one space, and there does not seem to have an easy way of coordinating operations across multiple environments or users, making it inadequate as a more generalized system.

one.world [12] is a system architecture developed at the University of Washington, designed to provide programmers with services for writing pervasive applications. The distributed components use remote event passing for communications, and perform discovery operations to locate resources for the components. The discovery server is centralized, but is elected in an ad-hoc manner from all the participating nodes, with the elections waited to favor candidates with the best response times. Once an election has taken place, all the nodes pass service information into the discovery server, and use this single point for most communication handling. one.world includes support for low-level resource allocation, but does not appear to have a generic way for modeling higher-level resources or handling conflicts. Because all components are also sharing the discovery server, there is a requirement for participating components to share information so that they can be discovered properly.

Chapter 10

Summary and Conclusion

I have shown that a system based on the four pillars:

1. Agent communications and structures that mirror the social and physical interactions they represent. See Chapter 4.

2. A collection of user preferences, mapped into utility functions, being used to fuel the environment's decisions about service mapping and resource management. This was described in Chapter 5.

3. An "awareness" of the state of the world, which can be used to further drive context-based preference decisions. Context-based preferences were further explored in Section 5.1.6.

4. A semantic memory that defines and describes the resource and preference requirements. The semantic network used to build this was described in Chapter 6.

will effectively serve in creating a system that will respect privacy, respect people's rights to control their own resources, adapt to changing environmental conditions, and handle individual preferences.

In order to show this, I have designed and built an augmented agent infrastructure for intelligent environments, called Hyperglue, based on this infrastructure.

During the design phase for these components, a focus was placed on creating infrastructure that serves the overall goal of building agents that can easily be assembled and added to a working system. This was one of Metaglue's most enduring design features, and impacting it as little as possible was a necessity.

To achieve this, I created subsystems that attempted to maximize a number of attributes. Perhaps foremost amongst these was flexibility, since the design needed to be easily extended beyond a handful of simple scenarios. This drove the creation of a knowledge representation that could be easily utilized and extended by the creation of new Java classes, and in turn a context model that could be easily adapted to new contextual cues.

I also created a model for agent societies that carefully preserves individual ownership and control, developing a hierarchy that attempts to model the same interactions we use in the real world. This achieves several purposes, including simplicity of communication and easier modeling of complex interactions by using abstraction barriers that hide the details of the individual societies.

Finally, I included a system for using a user's preferences to drive the selection of resources. This enhances flexibility, and provides a clean model for the user to specify desired behavior without becoming overly complex.

Appendix A

Scheme Code for Service Mapping

A.1 Code for Utility Function Generation

```
;;; -*- Mode: Scheme; Syntax: Kawa  -*-

;;; A Node space is a mapping of feature sets onto numerically ordered
;;; nodes The original version of this used binary features exclusively,
;;; this is a redesign for features that take on 1 of n values (e.g.
;;; speed in {fast medium slow}).  Each feature is assigned a byte of
;;; enough bits to cover the number of possible values and then the
;;; values are assigned values from 1 to n within that space.  The total
;;; feature vector is then the concatenation of these individual bytes.
;;;
;;; The node space has these components:
;;; The alist of features and their possible values
;;;
;;; The forward-map: A hash-table mapping feature-name to a byte-specifier
;;; for that feature and the list of values in order
;;;
;;; An inverse-map for mapping the numerical encoding to feature and value
;;; the inverse-map is an set of triples
;;; byte-specifier feature-name feature-values
;;;
;;; A vector of nodes, each corresponding to an element of the powerset
;;; of individual feature assignments.

;;; For example: if size is the second feature in bits 2 - 3
;;; encoding values large, medium small
;;; then large =  #o0100  ->  8
;;;      medium = #o1000  -> 16
;;;      small  = #o1100  -> 24

;;; note that the 00 case means no value, so the byte is always large
;;; enough to hold the number of values + 1.

;;; programming notes:
;;; (ldb (byte size position) number)
;;; (dpb newbyte (byte size position) number)
;;; are the common lisp ways of accesses bit-fields
;;; size and position are in bits

;;; Translations of defclasses

(require 'list-lib)

(define (nth i lst) (list-ref lst i))
(define (pushnew el lst) (if (member el lst) lst (cons el lst)))
```

```
(define-record-type netnode (make-netnode number outgoing incoming value)
  netnode?
  (number netnode-number)
  (outgoing netnode-outgoing set-outgoing!)
  (incoming netnode-incoming set-incoming!)
  (value netnode-value set-node-value!))

(define-record-type netlink (make-netlink incoming outgoing weight) netlink?
  (incoming netlink-incoming)
  (outgoing netlink-outgoing)
  (weight netlink-weight))

(define-record-type nodespace (make-nodespace fw-map inv-map node-vec feat)
          nodespace?
  (fw-map nodespace-forward-map)
  (inv-map nodespace-inverse-map)
  (node-vec nodespace-node-vector)
  (feat nodespace-features-list))

(define-record-type byte-specifier (make-byte-spec pos bits) byte-specifier?
  (pos byte-spec-position)
  (bits byte-spec-length))
(define (byte-spec-start lst) (byte-spec-position lst))
(define (byte-spec-end lst) (+ (byte-spec-position lst) (byte-spec-length lst)))

(define (make-hash-table) (make <java.util.HashMap>))
(define (ht-get key ht)
  (let ((value (invoke (as <java.util.HashMap> ht) 'get key)))
    (if (eq? value #!null) #f value)))
(define (ht-set! key value ht)
  (invoke (as <java.util.HashMap> ht) 'put key value))

;;; here the feature set is a set of (feature . values)
(define (make-node-space feature-set)
  (let ((forward-map (make-hash-table))
        (byte-position 0)
        (inverse-map '() ))

    (let ((mns-help (lambda (feature-name . feature-values)
                      (let* ((number-of-bits (integer-length
                                              (+ 1 (length feature-values))))
                             (byte-specifier
                              (make-byte-spec byte-position number-of-bits)))
                        (set! inverse-map
                              (cons (list byte-specifier feature-name
                                          feature-values)
                                    inverse-map))
                        (ht-set! feature-name
                                 (list byte-specifier feature-values)
                                 forward-map)
                        (set! byte-position
                              (+ byte-position number-of-bits))))))

      (for-each (lambda (f) (apply mns-help f)) feature-set)

      (let* ((network-size (arithmetic-shift 1 byte-position))
             (node-vector (make-array (shape 0 network-size))))
        (let loop ((i (- network-size 1)))
          (if (>= i 0)
              (let ((entry (make-netnode i '() '() #f)))
                (array-set! node-vector i entry)
                (loop (- i 1)))))
        (make-nodespace forward-map inverse-map node-vector feature-set)))))

;;; reminder:
;;; the inverse-map is an set of triples
;;; byte-specifier feature-name feature-values
```

```
(define (decode-node node ns)
  (let ((inverse-map (nodespace-inverse-map ns))
        (number (netnode-number node)))
    (filter (lambda (x) x)
            (map (lambda (ent)
                   (let* ((byte-spec (first ent))
                          (feature-name (second ent))
                          (feature-values (third ent))
                          (feature-number
                           (bit-extract number
                                        (byte-spec-start byte-spec)
                                        (byte-spec-end byte-spec))))
                     (if (and (> feature-number 0)
                              (<= feature-number (length feature-values)))
                         (list feature-name
                               (nth (- feature-number 1) feature-values))
                         #f)))
                 inverse-map))))

(define (deconstruct-nodespace ns)
  (let ((nv (nodespace-node-vector ns)))
    (list (nodespace-forward-map ns) (nodespace-inverse-map ns)
          (map (lambda (n) (decode-node n ns))
               (let lp ((i (array-start nv 0)))
                 (if (>= i (array-end nv 0)) '()
                     (cons (array-ref nv i) (lp (+ 1 i))))))
          (nodespace-features-list ns))))



;;; Given a set of feature-name feature-value pairs find the designated
;;; node

(define (dpb newbyte byte-spec integer)
  (let* ((start (byte-spec-start byte-spec))
         (end (byte-spec-end byte-spec))
         (lo (bit-extract integer 0 start))
         (hi (bit-extract integer end (integer-length integer))))
    (logior (arithmetic-shift hi end)
            (logior (arithmetic-shift newbyte start)
                    lo))))

(define (find-feature-set-node feature-set ns)
  (let ((node-vector (nodespace-node-vector ns))
        (key 0))
    (for-each (lambda (ent)
                (let ((feature-name (nth 0 ent))
                      (feature-value (nth 1 ent)))
                  (let-values (((byte-spec feature-value-number)
                                (map-feature-name-and-value
                                 feature-name
                                 feature-value
                                 ns)))
                              (set! key (dpb feature-value-number
                                             byte-spec
                                             key)))))
              feature-set)
    (array-ref node-vector key)))

(define (index-of tst lst)
  (cond ((null? lst) '())
        ((eq? tst (car lst)) 0)
        (else (let ((res (index-of tst (cdr lst))))
                (if (null? res) res (+ 1 res))))))

(define (map-feature-name-and-value feature-name feature-value ns)
  (let ((forward-map (nodespace-forward-map ns)))
    (let* ((feature-entry (or (ht-get feature-name forward-map)
                        (error "Invalid feature-name ~a" feature-name)))
          (byte-spec (nth 0 feature-entry))
          (feature-values (nth 1 feature-entry))
          (value-number (or (index-of feature-value feature-values)
                            (error "Invalid value ~a for feature ~a"
                                   feature-value feature-name))))
     (values byte-spec (+ 1 value-number)))))

;;; find all nodes matching a description and apply a function to
;;; each.  a description is a list of literals each literal is a list
;;; of feature-name and a specific-feature-value
;;;
;;; variable features are the feature-names for which we don't care
;;; what value they take on, we find
;;; all nodes with any value for this feature-name

(define (find-feature-set-nodes feature-set variable-features ns continuation)
  (let ((fixed-key 0))
    (for-each (lambda (ent)
                (let ((feature-name (nth 0 ent))
                      (feature-value (nth 1 ent)))
                  (let-values
                   (((byte-specifier feature-value-number)
                     (map-feature-name-and-value feature-name
                                                 feature-value ns)))
                   (set! fixed-key (dpb feature-value-number
                                        byte-specifier
                                        fixed-key)))))
              feature-set)

    (let do-one-more-dont-care ((remaining-features variable-features)
                                (key-so-far fixed-key)
                                (accumulated-dont-cares '()))
      (if (null? remaining-features)
          (continuation (array-ref (nodespace-node-vector ns) key-so-far)
                        accumulated-dont-cares)
          (let ((next-feature-name (car remaining-features))
                (remaining-features (cdr remaining-features)))
            (let* ((feature-entry
                    (or (ht-get next-feature-name (nodespace-forward-map ns))
                        (error "Invalid feature-name ~a" next-feature-name)))
                   (byte-specifier (first feature-entry))
                   (feature-values (second feature-entry)))
              (let lp ((feature-value (car feature-values))
                       (feature-values (cdr feature-values))
                       (i 1))
                (do-one-more-dont-care remaining-features
                                       (dpb i byte-specifier key-so-far)
                                       `((,next-feature-name ,feature-value)
                                         ,@accumulated-dont-cares))
                (if (not (null? feature-values))
                    (lp (car feature-values) (cdr feature-values)
                        (+ 1 i))))))))))

;;; This is used by canonicalize-rule below to combine the negative of
;;; the lhs of a rule with the rhs of the rule and vice-versa.  By
;;; negative we mean all possible features other than the one actually
;;; specified e.g. if the right hand side say (quality low) then we
;;; need to add to the left hand side (quality high) and (quality
;;; medium)

;;; so if we have (speed fast) > (quality low)
;;; where speed takes on values fast slow
;;; and quality takes on values high medium low
;;; then what we mean is
;;; (speed fast) & (quality high) > (speed slow) & (quality low)
;;; (speed fast) & (quality medium) > (speed slow) & (quality low)
```

```
(define (distribute and-term and-term-to-negate ns)
  (let ((answers '() ))
    (let do-all-remaining-dont-cares ((dont-care-terms and-term-to-negate)
                                      (stuff-so-far and-term))
      (if (null? dont-care-terms)
          (set! answers (pushnew stuff-so-far answers))
          (let ((feature-name (car (car dont-care-terms)))
                (feature-value (cadr (car dont-care-terms)))
                (dont-care-terms (cdr dont-care-terms)))
            (let ((feature-values (cdr (assoc feature-name
                                              (nodespace-features-list ns)))))
              (for-each (lambda (value)
                          (unless (eq? value feature-value)
                                  (let* ((new-term (list feature-name value))
                                         (accumulated-stuff
                                          (if (member new-term stuff-so-far)
                                              stuff-so-far
                                              (cons new-term stuff-so-far))))
                                    (do-all-remaining-dont-cares
                                     dont-care-terms accumulated-stuff))))
                        feature-values)))))
    answers))

; Take a rule and turn it into a canonical form.

(define (canonicalize-rule lhs rhs ns)
  (let ((completed-left (distribute lhs rhs ns))
        (completed-right (distribute rhs lhs ns))
        (answers '()))
    (for-each (lambda (l)
                (for-each (lambda (r)
                            (let ((ent (list l r)))
                              (set! answers (pushnew ent answers))))
                          completed-right))
              completed-left)
    answers))


;(canonicalize-rule '((speed fast)) '((quality low)) ns)
;(canonicalize-rule '((speed fast)) '((speed slow)) ns)

(define (process-a-rule better-features worse-features weight ns)
  (let ((used-terms '()) (dont-cares '()) (used-features '()))

    (define (do-symbols-in-thing conjunction)
      (for-each (lambda (term)
                  (set! used-terms (pushnew term used-terms))
                  (set! used-features (pushnew (car term) used-features)))
                conjunction))

    (do-symbols-in-thing better-features)
    (do-symbols-in-thing worse-features)
    (set! dont-cares (lset-difference equal?
                                      (map car (nodespace-features-list ns))
                                      used-features))

    (define (dominate-nodes better worse)
      (find-feature-set-nodes
       better dont-cares ns
       (lambda (better-node accumulated-dont-cares)
         (find-feature-set-nodes
          (append accumulated-dont-cares worse) '() ns
          (lambda (worse-node ignored)
            (let ((link (make-netlink better-node worse-node weight)))
              (set-outgoing! better-node
                             (pushnew link (netnode-outgoing better-node)))
              (set-incoming! worse-node
                             (pushnew link (netnode-incoming worse-node))))))))
    (let ((pairs (canonicalize-rule better-features worse-features ns)))
      (for-each (lambda (p)
                  (let ((better (first p))
                        (worse (second p)))
                    (dominate-nodes better worse)))
                pairs))))

(define (process-all-rules rules feature-set)
  (let ((node-space (make-node-space feature-set)))
    (for-each (lambda (rule)
                (let ((lhs (first rule))
                      (rhs (second rule)) (weight (third rule)))
                  (process-a-rule lhs rhs weight node-space)))
              rules)
    (assign-order node-space)
    node-space))


(define (assign-order node-space)
  (let ((node-vector (nodespace-node-vector node-space)))
    (define (do-one-node node path-so-far)

      (when (member node path-so-far)
            (error "Cycle ~{~%~a~^,~}"
                   (map (lambda (bad-node) (decode-node bad-node node-space))
                        (cons node path-so-far) )))
      (unless (netnode-value node)
              (let* ((best-of-descendants
                      (let ((maxval 0))
                        (for-each (lambda (outgoing-link)
                                    (let* ((descendant
                                            (netlink-outgoing outgoing-link))
                                           (weight
                                            (netlink-weight outgoing-link)))
                                      (do-one-node descendant
                                                   (cons node path-so-far))
                                      (let ((w (* weight
                                                  (netnode-value descendant))))
                                        (if (> w maxval) (set! maxval w)))))
                                  (netnode-outgoing node))
                        maxval))

                     ;; if there are no outgoing links the value is 1
                     (my-value  (if (zero? best-of-descendants) 1
                                    best-of-descendants)))
                (set-node-value! node my-value))))
    (let looper ((i (array-start node-vector 0)))
      (if (< i (array-end node-vector 0))
          (begin (do-one-node (array-ref node-vector i) '())
                 (looper (+ 1 i)))))))

(define (utility feature-set node-space)
  (let ((node (find-feature-set-node feature-set node-space)))
    (netnode-value node)))

(define (convert-surface-rule constraint)
  (display (list 'c-s-r constraint))
  (let ((pos (list-index list? constraint)))
    (let ((lhs (take constraint pos))
          (rhs (drop constraint (+ 1 pos)))
          (weight (second (nth pos constraint))))
      (define (group-it plist)
        (cond ((null? plist) '())
              (else (cons (list (first plist) (second plist))
                          (group-it (cddr plist))))))
      (list (group-it lhs) (group-it rhs) weight))))
```

```
(define (make-utility-function feature-set preferences)
  (let ((constraints (map convert-surface-rule preferences)))
    (let ((his-ns (process-all-rules constraints feature-set)))
      (lambda (features) (utility features his-ns)))))

(define-syntax defpreference
  (syntax-rules ()
    ((defpreference name constraint)
     (define name (convert-surface-rule (quote constraint))))))

;;;;; Testing

(define my-ns #f)
(define util-func1
  (make-utility-function
   '((speed fast slow) (quality high medium low) (privacy private public))
   '((speed fast (>> 2) speed slow)
     (quality high (>> 2) quality medium)
     (quality medium (>> 2) quality low)
     (privacy private (>> 2) privacy public)
     (speed fast (>> 2) privacy private)
     (quality medium speed fast (>> 2) privacy private)
     (privacy private speed fast (>> 2) quality low))))

;(util-func1 '((speed fast) (quality high) (privacy private)))
```

A.2 Additional Functions for Utility Bounds Calculations

```
;;; Utility Bounds function.  This is essentially performing the same
;;; job as the earlier utility function, except we pass it a
;;; continuation that will properly calculate the maximum and minimum
;;; values.

(define (utility-bounds bound-features unbound-features node-space)
  (let ((min nil)
(max nil))
    (find-feature-set-nodes
       bound-features unbound-features node-space
       (lambda (node ignored)
 (let ((node-val (netnode-value node)))
   (when (or (null? min) (< node-value min))
 (set! min node-val))
   (when (or (null? max) (> node-value max))
 (set! max node-val)))))
    (cons min max)))

;;; Note that the bounds version of the resultant utility function
;;; doesn't just take the features as an argument; it needs to take
;;; both the bound and unbound features.  The front-end that calls
;;; this needs to take that into account.

(define (make-utility-bounds-function feature-set preferences)
  (let ((constraints (map convert-surface-rule preferences)))
    (let ((his-ns (process-all-rules constraints feature-set)))
      (lambda (bound-features unbound-features)
(let ((result (utility-bounds bound-features unbound-features his-ns)))
  (cons min max)))))
```

A.3 Hierarchical Task Planning

This code utilizes a framework for performing Prolog-style logic programming. An implementation of this framework can be found in Appendix B.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Find Best Method for Service
;;;
;;; This finds the best method taking into account:
;;;  * The benefit delivered
;;;  * The cost of the resources
;;;  * The possiblity that some resource isn't working and could cause
;;;    the method to fail
;;;
;;; The utility-function is supposed to be the type that returns [max
;;; min] bounds.  This is always applied to the the top-level
;;; parameter alist since this is the value of the whole solution
;;;
;;; This builds a cutoff continuation that is used to cut-off a
;;; sub-tree exploration if it's clearly worse than the best solution
;;; so far.  This is equivalent to its (max utility - resource cost)
;;; incurred so far is worse than the best, conditioned by the
;;; probability of failure and failure cost for the top-level method
;;;
;;; It also builds a continuation for when a total solution is
;;; reached.  This captures the best solution so far.
;;;
;;; The main recursion is a couple of functions below
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (find-best-method-for-service service-name utility-function
      #!optional
      feature-requirements other-parameters)
  (let ((best-value nil)
(best-method nil)
(alist (make-alist-for-service service-name feature-requirements)))

    (define (cutoff? top-method total-resource-cost joint-probability)
      ;; This returns #f if you should cutoff, #t if there's a
      ;; chance this could be better
      (or (null? best-value)
  (let ((best-possible
 (expected-net-benefit top-method total-resource-cost
       joint-probability alist
       utility-function)))
    (or (null? best-value)
(> best-possible best-value)))))

    (find-methods-for-service
       service-name alist other-parameters
       (lambda (total-resource-cost joint-probability method
    resources sub-services)
 (let ((tradeoff (expected-net-benefit method
       total-resource-cost
       joint-probability
       alist utility-function)))
   (when (or (null? best-value) (> tradeoff best-value))
 (set! best-value tradeoff
       best-method (list method resources sub-services)))))
       cutoff?
       0
       1)

    best-method))
```

```
(define (make-alist-for-service service-name feature-requirements)
  (let* ((features (get-service-features service-name)))
    (map (lambda (feature)
   (let ((requirement (assoc feature feature-requirements)))
     (if requirement requirement
 (list feature (make-unbound-logic-variable feature)))))
 features)))

;;; The assumption is that the utility function returns max-value.
;;; The expected-cost-of-failure is the method's cost of failure
;;; weighted by probability of failure, which is 1 - probability that
;;; all the resources work.  "expected-utility" is the utility
;;; weighted by the probability that all the resources work your pay
;;; for the resources in all cases so it's not weighted by probability

(define (expected-net-benefit method resource-cost
      joint-resource-probability alist
      utility-function)
  (let* ((expect-failure-cost (* (- 1 joint-resource-probability)
 (cost-of-failure method)))
 (expected-utility (* joint-resource-probability
      (utility-function alist))))
    (- expected-utility resource-cost expect-failure-cost)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Find Methods for service does the hierarchical search
;;; It iterates for each top level method for the service (this is the or node):
;;; * fetchings the resources
;;; * binding the top-level parameters in alis
;;; *  and fetching the service requirements (sub-goals)
;;; It then checks whether this solution can be ignored.
;;;
;;; The idea is that the utility function is a bounds function which
;;; will return a max utility over all unbound service-parameters and
;;; we can then calculate an expected-net-benefit on that as we pursue
;;; sub-services.  The number of resources can only go up (increasing
;;; cost) and the number of bound-parameters can only go up,
;;; tightening bounding (decreasing max utility).
;;;
;;; Thus is expected-net-benefit is less than the best found so far,
;;; it isn't worth pursuing this line (since we're already worse and
;;; we can only get worse).  It then iterates over the sub-services
;;; requirements (this is the and node), recursively calling itself
;;; for each.  This iteration is done by recursion because each call
;;; is a generator that calls the continuation for each possible solution.
;;;
;;; For each complete solution (resources and sub-methods) it calls
;;; the continuation.
;;;
;;; The cutoff function is called with the method name, resource-cost,
;;; joint-probability.  It has access to the utility function and the
;;; alist as captured closure variables, so it does little more than
;;; compute expected-net-benefit and threshold on that
;;;
;;; The continuation is called once for each solution with:
;;; * the cost-of-all-resources used by that method
;;; * the joint probability that the resources all work
;;; * the method name
;;; * the resources used directly by that method
;;; * the sub-services alist for this method (which is a set of
;;;    triples -- one for each sub-service requirement including the
;;;    method name, resources and sub-services)
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun find-methods-for-service (service-name alist other-parameters
 continuation
 evaluation-function
 resource-cost-so-far
 resource-probability-so-far
 ;; Top-method is null on the
 ;; top-level call but not at any
 ;; other time
 #!optional top-method)
  (unless other-parameters
    (set! other-parameters '?stuff)

  (ask '[method-for ,service-name ,alist ?my-method ?my-resources
,other-parameters ?sub-services]
       #'(lambda (bs)
   (declare (ignore bs))
   (labels
       ((do-next (cost-so-far probability-so-far
sub-services-left sub-services-done top-method)
  (destructuring-bind  (sub-service-name his-alist other-params)
      (first sub-services-left)
    (find-methods-for-service
     sub-service-name his-alist other-params
     #'(lambda (his-resource-cost his-probability
his-method-name his-resources his-sub-methods)
 (setq sub-services-done (cons (list
his-method-name his-resources his-sub-methods)
       sub-services-done)
       sub-services-left (cdr sub-services-left))
 (if (null sub-services-left)
     ;; then you've completed the goal tree
report the solution to caller
     (funcall continuation his-resource-cost his-probability
      ?my-method ?my-resources sub-services-done)
   ;; more sub-goals, keep going
   (do-next his-resource-cost his-probability
sub-services-left sub-services-done
    top-method)))
     evaluation-function
     cost-so-far
     probability-so-far
     top-method))))
     (loop for r in ?my-resources
 for (value probability) = (is-working r)
 unless value
 do (return-from find-methods-for-service (values))
 do (incf resource-cost-so-far (cost-of-resource r))
    (setq resource-probability-so-far (*
resource-probability-so-far probability)))
     (when (funcall evaluation-function top-method
resource-cost-so-far resource-probability-so-far)
       (do-next resource-cost-so-far
resource-probability-so-far ?sub-services nil
(or top-method ?my-method))))))
```

Appendix B

Scheme Code for Logic Programming

```
;;; -*- Mode: Scheme; Syntax: Kawa; -*-

;;{{{ Common Debugging Options

(define *rule-tracing* #f)
(define *debug* '(
  ;unify
  ;ask-data
  ;ask-rule
  ;other
  ))

;;}}}

;;{{{ Useful Procedures

(define (atom? x) (not (pair? x)))
(define (printout . L)
  (map (lambda (x) (display x)) L)
  (newline))
(define (substitute-symbols A T)
  (cond ((symbol? T)
 (let ((subst (assoc T A)))
   (if (not subst) T (cdr subst))))
((not (pair? T)) T)
(#t (cons (substitute-symbols A (car T))
  (substitute-symbols A (cdr T))))))

;;}}}

(define *ruleMatcher* :: <edu.mit.aire.util.semantic.RuleMatcher> *ruleMatcherInstance*)

;;{{{ Dealing With Variables

;;; A logic variable is represented by a symbol whose print-name begins
;;; with a question-mark.
(define (is-variable? x)
  (and (symbol? x) (char=? (string-ref (symbol->string x) 0) #\?)))

;;; Find all logic variables in a piece of list structure.
;;; Takes an optional initial set of logic variables.

(define (variables-in-thing thing answer)
  (cond ((is-variable? thing)
 (if (not (memq thing answer))
     (cons thing answer)
     answer))
((pair? thing)
 (let ((a (variables-in-thing (car thing) answer)))
   (variables-in-thing (cdr thing) a)))
(#t answer)))

;;; Dereferencing chases logic variable until it's either unbound or
;;; bound to something other than a logic variable.

;;; Rule 1: Always dereference logic variables before doing anything
;;; else with them.

(define make-binding cons)
(define bind-var car)
(define bind-val cdr)

(define (dereference variable environment)
  (define (looper last-value)
    (let ((binding (assoc last-value environment)))
      ;; if we were looking it up then it had to be a variable. But if
      ;; we didn't find it then it's unbound, so return it.
      (cond ((not binding) last-value)
    ;; We found a value which isn't another variable so stop
    ;; here.
    ((not (is-variable? (bind-val binding))) (bind-val binding))
    (#t (looper (bind-val binding) )))))
  (looper variable))

(define (dereference-all pattern env)
  (cond ((is-variable? pattern)
 (dereference pattern env))
((pair? pattern)
 (cons (dereference-all (car pattern) env)
       (dereference-all (cdr pattern) env)))
(#t pattern)))

;;}}}

;;{{{ Unification of Patterns

(define (make-unification bind success) (list 'unify bind success))
(define (success? u) (caddr u))
(define (get-bindings u) (cadr u))

(define (unify pattern goal bindings)
  (if (memq 'unify *debug*)
      (printout "unify " pattern " " goal " " bindings))
  (cond ((is-variable? pattern)
 (unify-variable pattern goal bindings))
((is-variable? goal)
 (unify-variable goal pattern bindings))
((equal? pattern goal) (make-unification bindings #t))
((atom? pattern) (make-unification '() #f))
((atom? goal) (make-unification '() #f))
(#t
 (let ((u (unify (car pattern) (car goal) bindings)))
   (if (success? u)
       (unify (cdr pattern) (cdr goal) (get-bindings u))
       (make-unification '() #f))))))

(define (unify-variable variable stuff bindings)
  (let ((variable-value (dereference variable bindings)))
    (if (is-variable? variable-value)
(make-unification (cons (make-binding variable stuff) bindings) #t)
(unify variable-value stuff bindings))))

;;}}}

;;{{{ Ask -- The Major Interface
```

```
;;; Ask pursues a goal, given a binding environment it calls a "Success
;;; Continuation" every time it succeeds in finding a solution to the
;;; goal.  It passes the success continuation the binding environment
;;; which unifies the goal to the answer.  Success continuations are
;;; functions of one argument: the binding-environment.
;;; This could be haired up to pass in more to the success continuation

;;; Rule 2: Succeed by calling the success continuation
;;;         Give up by returning to your caller

(define (ask goal environment continuation)
  (ask-data goal environment continuation)
  (ask-rules goal environment continuation)

  ;; could also include ask-questions if you'd like
  )

;;; Look for assertions that unify with the goal
;;; Pass the success continuation the bindings resulting from each.
(define (ask-data goal environment continuation)
  ;; here we ask the Storage mechanism (through RuleMatcher) for the
  ;; right stuff.
  (define (looper tuple-list)
    (if (not (null? tuple-list))
(let* ((assertion (car tuple-list))
       (extended-binding
(get-bindings (unify assertion goal environment))))
  (if (not (null? extended-binding))
      (continuation extended-binding))
  (looper (cdr tuple-list)))))

  ; this only works for tuples
  (if (= 3 (length goal))
      (begin
(if (memq 'ask-data *debug*)
    (printout "ask-data " goal " " environment))
(let* ((modified-goal (map (lambda (x)
     (if (is-variable? x)
 (let ((y (dereference x environment)))
   (if (is-variable? y) #!null
       y))
 x)) goal))
       (toss (if (memq 'other *debug*)
 (printout " mod-goal: " modified-goal)))
       (tuples (apply invoke
      (cons
       *ruleMatcher*
       (cons
'getMatches modified-goal)))))
  (if (memq 'other *debug*)
      (printout (length tuples) " matches for " goal))
  (looper tuples)))))

;;; Just for prettiness
(define (rule-name r) (car r))
(define (rule-if-part r) (cadr r))
(define (rule-then-part r) (caddr r))

(define (ask-rules goal environment continuation)
  (define (looper rule-list)
    (if (not (null? rule-list))
(begin (ask-rule (car rule-list) goal environment continuation)
       (looper (cdr rule-list)))))
  (looper (invoke *ruleMatcher* 'getRules)))

;;; Increment once each time you instantiate a rule so as to get new
;;; variable names.
(define *counter* 0)
```

```
;;; Given a set of variables (e.g. foo bar) build a corresponding set
;;; with new names (e.g. foo-n bar-n) and return an alist of new to old
;;; names.
(define (renaming-alist-for-variables variables)
  (define (looper var-list ret-list)
    (if (not (null? var-list))
(looper (cdr var-list)
(cons (cons (car var-list)
    (string->symbol (format #f "~A-~D" (car var-list)
    *counter*)))
      ret-list))
ret-list))
  (set! *counter* (+ 1 *counter*))
  (looper variables '()))

(define (renaming-alist-for-rule rule)
  (let ((variables (variables-in-thing
     (rule-if-part rule)
     (variables-in-thing (rule-then-part rule) '() ))))
    (renaming-alist-for-variables variables)))


;;; Ask Rule unifies the Then-Part of rule to the goal.  if successful,
;;; it then does an Ask-ANd of the If-part Before it does this, it
;;; renames all the variables.  It passes to Ask-And a success
;;; continuation which calls the original success continuation
;;; after optionally tracing the rule.

(define (ask-rule rule goal bindings continuation)
  (if (memq 'ask-rule *debug* )
      (printout "ask-rule " rule " "  goal " " bindings ))
  (let* ((renaming (renaming-alist-for-rule rule))
 (renamed-then-part (substitute-symbols renaming
(rule-then-part rule)) )
 (u (unify goal renamed-then-part bindings))
 (extended-bindings (get-bindings u))
 (success (success? u)))
    (if success
(begin
  (if *rule-tracing*
      (printout "Attempting Rule "
(rule-name rule) " on goal "
(substitute-symbols bindings goal)))
  (ask-and (substitute-symbols renaming (rule-if-part rule))
   extended-bindings
   (if *rule-tracing*
       (lambda (winning-bindings)
 (printout "Rule "
   (rule-name rule)
   " won "
   (substitute-symbols winning-bindings
       renamed-then-part))
 (continuation winning-bindings))
       continuation))
  (if *rule-tracing*
      (printout "Rule "
(rule-name rule)
" can do no more on goal "
(substitute-symbols bindings goal))))))

;;; Ask-and ASKs each sub-goal in turn passing in the bindings so far
(define (ask-and goals bindings continuation)
  (if (memq *debug* 'other)
      (printout " ask-and " goals " " bindings))
  (if (null? goals)
      (continuation bindings)
      (ask (car goals)
    bindings
    (lambda (extended-bindings)
    (ask-and (cdr goals)
     extended-bindings
     continuation)))))

;;}}}

;;{{{ Internal defrule and deftrigger

(define (defrule-internal name if then)
  (invoke *ruleMatcher* 'defineRule name if then))

; trigger definition -- this creates a procedure that matches a tuple
; to a set of rules, and triggers the body if there's a match.  Note
; that the formals are only matched against newly-generated tuples, so
; these are only triggered once for a new object.
(define (deftrigger-internal name formals patterns body)
  (invoke *ruleMatcher* 'defineTrigger name
  (lambda (test-tuple)
    (let* ((unification (unify (car formals) test-tuple '()))
   (success (success? unification))
   (new-bindings (get-bindings unification)))
      (if success
  (begin
    (ask-and patterns new-bindings
     (lambda (bindings)
       (eval (dereference-all body bindings)))))))))))
;;}}}
```

Bibliography

[1] Gregory Abowd, Barry Brummitt, and Steven Shafer, editors. *Ubicomp 2001*, volume LNCS 2201, Atlanta, GA, USA, 2001. Springer-Verlag.

[2] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *17th ACM Symposium on Operating Systems Principles (SOSP)*, Kiawah Island, SC, December 1999.

[3] MIT Project AIRE. Web page. http://www.ai.mit.edu/projects/aire/.

[4] Stavros Antifakos, Florian Michahelles, and Bernt Schiele. Proactive instructions for furniture assembly. In *UbiComp '02: Proceedings of the 4th International Conference on Ubiquitous Computing*, pages 351–360, London, UK, 2002. Springer-Verlag.

[5] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, May 2001.

[6] Per Bothner. Kawa, the Java-based Scheme system. Web page. http://www.gnu.org/software/kawa.

[7] Barry Brumitt, Brian Meyers, John Krumm, Amanda Kern, and Steven Shafer. EasyLiving: Technologies for intelligent environments. In Peter Thomas and Hans W. Gellersen, editors, *Handheld and Ubiquitous Computing: Second International Syposium*, pages 12–29, Bristol, UK, September 2000. HUC, Springer.

[8] Michael Coen, Brenton Phillips, Nimrod Warshawsky, Luke Weisman, Stephen Peters, and Peter Finin. Meeting the computational needs of intelli-

gent environments: The Metaglue system. In Paddy Nixon, Gerard Lacey, and Simon Dobson, editors, *Managing Interactions in Smart Environments: 1st International Workshop on Managing Interactions in Smart Environments (MANSE '99)*, pages 201–212. Springer-Verlag, London, 1999.

[9] John Ellson, Emden Gansner, Eleftherios Koutsofios, John Mocenigo, Stephen North, and Gordon Woodhull. Graphviz - open source graph drawing software. http://www.research.att.com/sw/tools/graphviz.

[10] Dieter Fensel, Ian Horrocks, Frank van Harmelen, Stefan Decker, Michael Erdmann, and Michel C. A. Klein. OIL in a nutshell. In *Knowledge Acquisition, Modeling and Management*, pages 1–16, 2000.

[11] Krzysztof Gajos. Rascal - a resource manager for multi agent systems in smart spaces. In *Proceedings of CEEMAS 2001*, 2001.

[12] Robert Grimm, Janet Davis, Eric Lemar, Adam MacBeth, Steven Swanson, Tom Anderson, Brian Bershad, Gaetano Borriello, Steven Gribble, and David Wetherall. Programming for pervasive computing environments. Technical Report UW-CSE-01-06-01, University of Washington, Department of Computer Science and Engineering, 2001.

[13] Arnt Gulbrandsen, Paul Vixie, and Levon Esibov. A DNS RR for specifying the location of services (DNS SRV). RFC 2782, February 2000.

[14] Robert Headon and Rupert Curwen. Recognizing movements from the ground reaction force. In *Workshop on Perceptive User Interfaces*. ACM Digital Library, November 2001. ISBN 1-58113-448-7.

[15] Boris Katz. Using English for indexing and retrieving. In P.H. Winston and S.A. Shellard, editors, *Artificial Intelligence at MIT: Expanding Frontiers*, volume 1. MIT Press, Cambridge, MA, 1990.

[16] Miryung Kim, Gary Look, and João Sousa. Planlet: Supporting plan-based user assistance. Technical Report IRS-TR-03-005, Intel Research Lab, Seattle, 2003.

[17] Max Van Kleek, Tyler Horton, and Elizabeth Boyle. SKINNI: Connecting co-workers using information kiosks in the workplace. http://www.ai.mit.edu/ emax/papers/CHI-Skinni-v6.pdf.

[18] Buddhika Kottahachchi. ACCESS: Access Controls for Cooperatively Enabled Smart Spaces. Masters of Engineering Thesis, Massachusetts Institute of Technology, Cambridge, MA, February 2005.

[19] Buddhika Kottahachchi and Robert Laddaga. Access controls for intelligent environments. In *Proceedings of ISDA '04: 4th Annual International Conference on Intelligent Systems Design and Applications*, Budapest, Hungary, August 2004.

[20] Ajay Kulkarni. Design principles of a reactive behavioral system for the Intelligent Room. *Bitstream: The MIT Journal of EECS Student Research*, 2002. To appear.

[21] Ray Kurzweil. The law of accelerating returns, March 2001.

[22] V. Lesser, M. Atighetchi, B. Benyo, R. Horling, V. Anita, W. Regis, P. Thomas, S. Xuan, and Z. Zhang. The intelligent home testbed. In *In Proceedings of the Autonomy Control Software Workshop (Autonomous Agent Workshop)*, 1999.

[23] Victor Lesser, Michael Atighetechi, Brett Benyo, Bryan Horling, Anita Raja, Régis Vincent, Thomas Wagner, Ping Xuan, and Shelly X.Q. Zhang. A multi-agent system for intelligent environment control. In *Proceedings of the Third International Conference on Autonomous Agents (Agents99)*, Seattle, WA, 1999.

[24] Jeremy Lilley. Scalability in an intentional naming system. Master's thesis, Massachusetts Institute of Technology, May 2000.

[25] David L. Martin, Adam J. Cheyer, and Douglas B. Moran. The Open Agent Architecture: A framework for building distributed software systems. *Applied Artificial Intelligence*, 13(1-2):91–128, January-March 1999.

[26] Michael McGeachie and Jon Doyle. Efficient utility functions for ceteris paribus preferences. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, Edmonton, Alberta, August 2002.

[27] Sun Microsystems. Jini network technology. Web Page. http://java.sun.com/products/jini/.

[28] Sun Microsystems. Jini lookup service specification v1.0. Web Page, 1999. http://www.sun.com/software/jini/specs/jini10specs/lookup-spec.html.

[29] Nelson Minar, Matthew Gray, Oliver Roup, Raffi Krikorian, and Pattie Maes. Hive: Distributed agents for networking things. In *Proceedings of ASA/MA'99, the First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents*, 1999.

[30] Marvin Minsky. *The Society of Mind*. Simon & Schuster, Inc., 1986.

[31] MySQL: The world's most popular open source database. Web page. http://www.mysql.com/.

[32] http://o2s.csail.mit.edu/.

[33] Alice Oh, Rattapoom Tuchinda, and Lin Wu. MeetingManager: A collaborative tool in the Intelligent Room. In *Student Oxygen Workshop*, Cambridge, MA, 2001. http://www.ai.mit.edu/projects/iroom/publications/alice-sow01.pdf.

[34] Brenton Phillips. Metaglue: A programming language for multi-agent systems. M.Eng thesis, MIT, 1999.

[35] Claudio Pinhanez. The Everywhere Displays projector: A device to create ubiquitous graphical interfaces. In Abowd et al. [1], pages 315–331.

[36] Shankar R. Ponnekanti, Brian Lee, Armando Fox, Pat Hanrahan, and Terry Winograd. ICrafter: A service framework for ubiquitous computing environments. In *Ubicomp*, Atlanta, Georgia, September 2001.

[37] PostgreSQL. Web page. http://postgresql.org/.

[38] M.R. Quillian. Semantic memory. In M. Minsky, editor, *Semantic Information Processing*, pages 227–270. MIT Press, Cambridge, MA, 1968.

[39] Cliff Randell and Henk Muller. Low cost indoor positioning system. In Abowd et al. [1], pages 42–48.

[40] Michael K. Smith, Chris Welty, and Deborah L. McGuinness. Owl web ontology language guide, February 2004. W3C Recommendation.

[41] TouchGraph development page. Web page. http://touchgraph.sourceforge.net/.

[42] John Veizades, Erik Guttman, Charles E. Perkins, and Scott Kaplan. Service location protocol. RFC 2165, June 1997.

[43] Nimrod Warshawsky. Extending the Metaglue multi-agent system. M.Eng thesis, MIT, Cambridge, MA, 1999.

[44] Mark Weiser. The computer for the 21st century. *Scientific American*, 265(3):66–75, January 1991.

[45] Weikai Xie, Yuanchun Shi, and Guanyou Xu. Smart Classroom – an intelligent environment for tele-education. In *Proceedings of the Second Pacific-Rim Conference on Multimedia*, Beijing, China, 2001.

[46] Weikai Xie, Yuanchun Shi, Guanyou Xu, and Yuanhua Mao. Smart Platform – a software infrastructure for smart space (SISS). In Submission.