Metaglue: A Programming Language for Multi-Agent Systems

by

Brenton A. Phillips

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical Engineering and Computer Science

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

January 20, 1999

[February 1999]

© Copyright 1999 Brenton A. Phillips. All rights reserved.

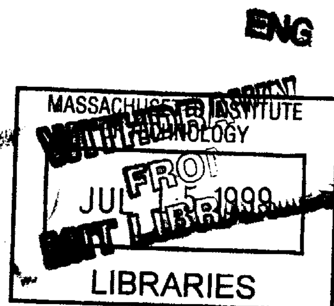
The author hereby grants to M.I.T. permission to reproduce and distribute publicly paper and electronic copies of this thesis and to grant others the right to do so.

Author
Department of Electrical Engineering and Computer Science
January 20, 1999

Certified by
Tomás Lozano-Pérez
Thesis Supervisor

Accepted by
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

ENG



Metaglue: A Programming Language
for Multi-Agent Systems
by
Brenton A. Phillips

Submitted to the
Department of Electrical Engineering and Computer Science

January 20, 1999

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer Science and Electrical Engineering
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Metaglue is an extension to the Java programming language that provides very high-level support for writing groups of small software agents that interact with one another. Metaglue was developed as part of the AI Lab's Intelligent Room Project. The Intelligent Room has literally dozens of hardware and software components that run on a variety of networked workstations. We needed a system that could link all of these components and coordinate the flow of data among them.

The computational needs of the Intelligent Room -- while not unique -- were not satisfied by any pre-existing software systems or programming environments. We wanted the Intelligent Room's software infrastructure to be persistent, robust, and dynamically reconfigurable. We needed the ability to modify (or even introduce) individual components without bringing the whole system down, and we wanted to have tools for understanding and debugging the behavior of large groups of interacting software agents.

Thesis Supervisor: Tomás Lozano-Pérez
Title: Associate Head, Computer Science

TABLE OF CONTENTS

1 Introduction....................7

1.1 Target Application....................8

1.2 Software Agents....................10

1.3 The Design Goals of Metaglue....................12

1.3.1 Linking Agents Together....................13

1.3.2 Establishing and Maintaining Agent Requirements....................15

1.3.3 Introducing and Modifying Agents in a Running System....................18

1.3.4 Debugging a Running Agent System....................21

1.4 A Brief History of Metaglue....................22

2 Metaglue Overview....................25

2.1 A Single Agent....................26

2.2 The *tiedTo* Primitive....................27

2.3 Spreading....................28

2.4 Adding Another Agent....................29

2.5 The *reliesOn* Primitive....................30

2.6 Reconnection....................33

2.7 Agent Swapping....................34

2.8 Attributes....................34

2.9 An Extended Example....................36

3 How Metaglue Works....................49

3.1 Metaglue Agents....................49

3.1.1 Agent Naming Scheme: AgentIDs....................49

3.1.1.1 Societies....................50

3.1.1.2 Occupations....................51

3.1.1.3 Designations....................51

3.1.1.4 Nomenclature....................52

3.1.2 Agent Definitions....................53

3.1.2.1 The *reliesOn* Primitive....53
3.1.2.2 The *tiedTo* Primitive....57
3.1.2.3 The Attribute Primitives....58
3.1.2.4 The Naming Primitives....60
3.1.2.5 The *shutdown* Primitive....60
3.2 Metaglue System Agents....61
3.2.1 The Agent Agent....61
3.2.2 The Metaglue Manager Agent....62
3.2.2.1 The *findAgent* Method....62
3.2.2.2 The *startAgent* Method....62
3.2.2.3 The *shutdownAgent* Method....63
3.2.3 The Catalog Agent....64
3.2.3.1 The *lookup* Method....64
3.2.3.2 The Catalog Monitor Agent....65
3.2.4 The Attribute Manager Agent and the Configurator Agent....66
3.3 The Metaglue Virtual Machine and Spreading....66
3.3.1 The MVM....66
3.3.2 The Glue Spreader....67
3.4 Keeping It All Alive....68
3.4.1 Agent Reconnection....68
3.4.2 Agent Swapping....69
3.5 Debugging a System of Agents....70

4 Conclusions....77
4.1 Report Card....77
4.1.1 Meeting the Goals....77
4.1.2 Limitations....79
4.2 Challenges....80
4.3 The Take Home Lesson....81
4.4 Future Work....81

5 Bibliography......83

LIST OF FIGURES

2.1 VCR Agent......41

2.2 *TiedTo*......42

2.3 Spreading......43

2.4 Speech Agent......44

2.5 IR Agent......45

2.6 Couch Scenario......46

2.7 Command Post......47

3.1 Configurator......73

3.2 Catalog Monitor......74

3.3 Agent Definition......74

3.4 Agent Tester......75

ACKNOWLEDGMENTS

I thank my father and mother, John and Kathy Phillips, for their supporting me in everything I've ever done, including this thesis. Indeed, my whole family has been supportive of me in this. My sister Heather and cousin Bo Boatright always encourage me.

I thank my friends for helping me work and play – Tim Cargol, Elizabeth Dale, Lesley Ford, Tim McAnaney, Mary Obelnicki, Owen Ozier, and Corissa Thompson.

I thank my friends in the Intelligent Room project – Krzysztof Gajos, Marion Groh, Josh Kramer, Stephen Peters, Nimrod Warshawsky, Luke Weisman, and Kevin Wilson. We always have a great time working together.

I especially thank Michael Coen. He brought me up in the AI Lab, and taught me nearly everything I know about AI. I also thank my thesis supervisor, Tomás Lozano-Pérez, for giving me freedom in my direction on this project.

This material is based upon work supported by the Advanced Research Projects Agency of the Department of Defense under contract number F30602-94-C-0204, monitored through Rome Laboratory.

Lastly, I dedicate this thesis to my grandmother, Harriette Boatright. She has been my best buddy since I was born.

Metaglue: A Programming Language for Multi-Agent Systems

CHAPTER 1

Introduction

Traditional programming languages provide no support for managing systems of interactive, distributed computations, i.e., those in which different components run asynchronously on a heterogeneous collection of networked computers. Metaglue is an extension to the Java programming language that provides high level support for writing groups of interacting software components, called agents. Metaglue automates the connection and distribution of these agents according to their computational requirements. Metaglue also maintains the connections between agents and allows for agents to be removed, modified, and replaced in a running system.

Metaglue extends Java by introducing a new agent class that serves as the ancestor of all Metaglue agents. By writing agents that extend this agent class, agents can make use of methods (the Metaglue primitives) for controlling their behavior. Metaglue also includes a runtime platform, the *Metaglue Virtual Machine*, on which all agents run. There are few special Metaglue system agents that establish, distribute, and connect agents.

In this thesis, we will investigate the design and implementation of Metaglue in the context of its primary target application, the Intelligent Room project of the Human-

Computer Interaction Group in the MIT Artificial Intelligence Laboratory. However, we will argue that applications other than the Intelligent Room would benefit from using Metaglue - the computational demands that Metaglue satisfies in this application are not unique. However, since Metaglue is actively being used in the Intelligent Room on a daily basis, we will draw our examples and motivations for Metaglue from its primary application. We begin with a description of Hal, one of the research efforts of the Intelligent Room project.

1.1 Target Application

Hal is a highly interactive space (in room 835 of the MIT AI Lab) in which computation is invisibly and seamlessly used to enhance ordinary activity[1]. This room watches its occupants with cameras and listen to them with microphones. Hal then intelligently assists the occupants with their work, organization, communication, and relaxation by responding to and even anticipating their needs.

For example, if someone lies down on the couch to nap in Hal, Hal will close the blinds and drapes and dim the lights. It can also turn on some soft music according to the preferences of the occupant, take messages for incoming phone calls, and tape any television programs that the occupant would normally watch during this time. Hal will also ask the occupant when they wish to be awakened, and the person simply responds by speaking out loud.

[1] I will be using the name "Hal" to refer to both the physical room and the embedded computation system in the room. The intended meaning should be clear from the context.

Another example is a command-post style interaction. A person can request to see a map. The blinds, drapes, and room lighting are adjusted so that the projected map display is easily visible on a wall. Using a laser pointer, the person can then annotate and indicate on the map. The person might mark out a region on the map, the Middle East for example, and tell Hal to zoom in on this region of the world. Then, the user might indicate Baghdad and ask for information about this city. Hal will project a web browser on the wall with a page showing information about Baghdad.

It is easy to imagine more complex scenarios in running in Hal, but just from these simple interactions, one can see a need for orchestrating the data flow between a diverse set of subsystems (vision, speech production/recognition, lighting, etc…) that have a different set of computational requirements (see figure 2.6). Hal has literally dozens of hardware and software components that run on a variety of networked workstations (IRIX, Solaris, SunOS, Linux, and Windows 95/NT). We needed a system for Hal that could provide the *computational glue* for linking all of these components and coordinating the flow of data among them.

The computational needs of Hal, while not unique, were not satisfied by any preexisting software systems or programming environments. We wanted Hal's software infrastructure to be persistent, robust, and dynamically reconfigurable. We needed the ability to introduce and modify individual components without bringing the whole system down, and we wanted to have tools for understanding and debugging the behavior of large groups of interacting components.

Metaglue was designed to meet these needs. Using Metaglue, we have written a robust, distributed controller for Hal that consists of more than eighty software components running on ten workstations.

While building the software control system for Hal using Metaglue, we learned some "take home" lessons that will help in designing future agent programming languages so that they are not imposing to programmers and prove to be useful development tools. Agent programming languages should remain simple, i.e. easy to learn and easy to remember. They should provide a focused set of primitives for managing the software components of the system and avoid the temptations of *creeping featurism* – the tendency to add every new capability that comes to mind during development. The key objective here is ease-of-use. By keeping Metaglue simple and focused, Metaglue makes it easy to convert pre-existing software components into *Metaglue agents.* It also makes Metaglue an attractive platform for developing systems of distributed, interacting, and persistent agent systems. There is more about what we learned from designing and using Metaglue in the conclusions.

In the rest of this introduction, we will define the software component building blocks of Metaglue, called agents. This is followed by a discussion of the design goals of Metaglue, their motivations, and the criteria used to evaluate how successfully Metaglue meets these goals. We conclude the introduction with a short history of Metaglue.

1.2 Software Agents

We have defined Metaglue as a system for expressing, integrating, and managing software agents. But what exactly are software agents? The term "software agent" is not uniformly defined among the people who use it [Sycara]. It generally refers to a program that acts on behalf of a human in order to perform laborious information management tasks. This sometimes involves moving among hosts and often involves intelligently interacting with other programs (e.g. agents) and people. Agents sometimes adapt over time to the needs of their human user and the shape of the information space that they manage information in. The term "software agent" often refers more specifically to a program that represents a person in electronic markets, filters and presents information for the person, or acts as personalized secretary.

Agents are often characterized as autonomous, intelligent, and robust. The realization of these traits generally requires much complexity, leading to large, monolithic software components. This centralization is problematic for many reasons: it creates a bottleneck in resource access, it produces a single point of failure, and it makes it difficult to reuse particular agent characteristics in other contexts. [Coen97]

An alternative approach to designing agents is characterized by what are known as multi-agent systems (MASs), for which Metaglue is designed. Agents in a MAS perform individually specialized, simple tasks, but they connect together in a web of intercommunication to cooperatively perform complex activities. Metaglue agents tend to be small in terms of code because they do not perform the more general tasks of monolithic agents. If fact, there is a generally accepted one-page rule of thumb for agent code among the agent programmers in Hal. Most Metaglue agents in Hal actually just

wrap other software components and simply add the Metaglue primitives to express the requirements for those components.

The use of the term "agents" in the remainder of this paper refer to the simple, intercommunicating, focused-task agents used in MASs and not monolithic agents.

1.3 The Design Goals of Metaglue

The need for a control system meeting the computational demands of Hal motivated Metaglue. A system of agents that would accomplish the required coordination illustrated in the scenario above is one of a more general class of agent systems. This class includes agent systems that are distributed across a network of heterogeneous hardware and software environments, robust to failures of individual subsystems, capable of continually maintaining a well-defined set of services, and expandable/modifiable during runtime. We will justify these requirements in the discussion below.

In order to support these computational requirements and create a practically useable agent development system, we established four design goals for Metaglue:

1. Establish communication channels between software components that may or may not have been designed to explicitly cooperate.
2. Establish and maintain the configuration that each agent specifies in its requirements for operation.

3. Permit the introduction and modification of agents without taking the whole system down.
4. Support the debugging of a running system of agents.

In the following sections, we will explain and motivate these goals.

1.3.1 Linking Agents Together

Metaglue is designed to provide *computational glue* for integrating disparate agents. Computational glue is the establishment of paths of communication between individual software components that use one another's services even when those components were not designed to explicitly work with each other. Providing computational glue for integrating different kinds of agents is motivated by the fact that there are applications such as Hal that face three engineering challenges: they are complex, they are computationally intensive, and their components were not designed to work together.

The first of these challenges is that applications such as Hal are large and complex. This complexity manifests itself in terms of the large amount of code required both to connect subsystems together and to coordinate the flow of information throughout the whole system. Other intelligent environment applications, such as the Kids Room at the MIT Media Laboratory, admit to the same need for very large amounts of control code (emphasis added):

> "For the room to adequately model and respond to all of these scenarios it will require both especially clever story design and *tremendous amounts of narration and control code*. The more person-object interactions that the system fails to handle in a natural way, the less immersive and sentient the entire system feels." [Bobick]

The second challenge is that applications such as Hal and the Kids Room need large amounts of computation that is not available on an individual computer. In other words, they are not practically realizable without being distributed. In Hal, each vision and speech recognition system consumes the resources of an entire workstation in order to provide real-time service. This is reflected in the "responsiveness" requirement mentioned in the quote above. This issue is not simply an artifact of current limitations in CPU speeds and is not likely to become irrelevant any time soon. Because many of the subsystems are usually developed in isolation, their designers will surely continue to take advantage of any upcoming technology improvements that result in faster processing speeds. Furthermore, since many of these systems, e.g., speech recognition, are performing progressive real-time searches, they can naturally generalize to consume arbitrary increases in available computational power.

The third and final engineering challenge that applications such as Hal face is that some applications are built out of components that were not designed to explicitly cooperate. One example comes from another project in the MIT AI Laboratory. Boris Katz's START team researches natural language database interactions. They wanted to interact with the START system using speech for input and browser display for output. This requires that speech recognition software (IBM ViaVoice) and a web-browser (Netscape) cooperate with the START engine itself. None of these systems have any knowledge of or any easy way to connect with the others. The START team did not have

the resources or interest in spending the time to integrate these components, and such an ad hoc integration would be difficult to maintain. A control system to link these heterogeneous components together was needed.

1.3.2 Establishing and Maintaining Agent Requirements

Metaglue is designed to establish and attempt to maintain the configuration that each agent specifies in its requirements for performing tasks. Agents have external needs, such as what hardware they need access to. They also specifically require other agents to help them accomplish their tasks. For example, if one agent relies on a second agent in a running agent system, the second agent needs to exist and be functional so that the first agent can provide its services. The second agent may also rely on yet a third agent, and so forth, resulting in a complex web of agent dependency.

This web of agent reliance quickly gets very complicated as the agents assemble an increasingly complex dependency topology. The couch scenario in Hal described above is a fairly limited interaction but requires the coordinated interaction of over eighty software agents for proper operation. This motivates the goal of Metaglue to automatically establish a system of agents and their interconnections.

Not only should Metaglue establish the agents and their interconnections, but also it should attempt to maintain them. In the fairly simple Hal couch scenario, the eighty or so agents are distributed across approximately ten workstations. If connections between these agents break, the scenario breaks down. There are three primary reasons that connections are severed between agents:

1. Networks may be flaky, hosts may crash or be rebooted, and buggy agents may crash or be killed.
2. Agents are stopped and removed for modification.
3. Agents move.

First, the network connecting the workstations can be flaky. This is becoming rare as local networking systems improve and has become increasingly irrelevant in Hal. However, network problems still remain relevant in more widely distributed software systems such as the Internet and wireless mobile networks. Also, connections between agents are broken when the workstations that agents are running on crash or buggy agents fail.

During such failures, Metaglue compensates for the lost or inaccessible services of necessary agents so that the system of agents doesn't completely break. This may involve Metaglue trying to fix the severed connections between agents, or it may involve restarting needed agents somewhere else and reestablishing connections to the newly started agents.

The fact that Metaglue maintains connections despite network, host, and agent failures is a side effect of the reconnection mechanism rather than a motivation for this mechanism. It is not a goal of Metaglue to solve these problems. Agents being stopped for modification and agents moving from host to host are the real motivations for Metaglue maintaining connections. We discuss these motivations now.

A second and a more common reason agent connections are broken and need to be restored is that an agent is explicitly stopped so that it can be modified. Agents mature as their abilities are enhanced and their methods of accomplishing their tasks are altered. When an agent is removed from the system to be modified, its connections between other agents are broken[2]. When the agent is restarted, Metaglue automatically restores the connections (see section 3.4.1 for more on reconnection).

Finally, the fact that agents can move from host to host motivated the goal of maintaining the connections between agents. When agents move, their connections do too. Agents may move to satisfy a special hardware or software need. They may also move to improve performance (i.e. to a workstation with more computational resources or to gain local access to other needed agents).

In the Hal couch scenario described in section 1.1, the agents that deal with the vision systems detecting a person laying on the couch run only on hosts with frame-grabber hardware. The speech recognition agent needs to run on a host where the speech recognition software runs (IBM ViaVoice is available only on Microsoft Windows platforms). These agents can be launched from a single host that does not satisfy the requirements of the agents. In Hal with more than eighty agents operating across ten workstations, it would be very tedious to remember which agents should be launched from where. Not only that, but the workstations may be reconfigured, i.e. hardware and software may be moved. Metaglue determines the specific requirements of each agent (other agents, hardware, software, etc...) and will move the agents to satisfy these needs

[2] Metaglue currently provides the basic mechanism for the removal and reinsertion of agents. The semantics of this feature are being worked out in an M. Eng. thesis by another Intelligent Room student.

as necessary. When an agent moves across hosts, its connections have to be restored, which Metaglue does automatically.

Agents may also need to move for load-balancing purposes. Imagine a load-balancing agent that analyzes a running system of agents and tries to distribute the computational requirements of a system of agents across hosts. A load-balancing agent may recommend that some agents move from a host because some other agents on the host need exclusive access to the host's computational resources. It may also recommend that agents move from a host because the host has become overloaded with other activities or because parts of the local network are experiencing heavy traffic and are limiting access to important agents. Metaglue must be able to take the recommendations of the load-balancing agent and move agents to new hosts that satisfy the hardware and software requirements of the agents. Again, essential to the success of this moving of agents is the fact that Metaglue must restore the necessary interconnections between them.

In summary, Metaglue automatically distributes and connects agents. It also reestablishes connections severed when agents are stopped for modification or when agents move.

1.3.3 Introducing and Modifying Agents in a Running System

Metaglue is designed to permit the introduction and modification of agents without taking the whole system down. The need for introducing new agents into a running system of agents was anticipated early on for systems of agents controlling

spaces such as Hal. In 1991, Mark Weiser at Xerox PARC predicted the need for operating environments that would allow components to be added and removed from a running system controlling such environments:

> "Today's operating systems, like DOS and UNIX, assume a relatively fixed configuration of hardware and software at their core...But in an embodied virtuality [a computer seamlessly integrated into the environment], local devices come and go, and depend upon the room and the people in it. New software for new devices may be needed at any time, and you'll never be able to shut off everything in the room at once. Future operating systems based around tiny kernels of functionality may automatically shrink and grow to fit the dynamically changing needs of ubiquitous computing."
> [Weiser]

New agents may be introduced to control newly introduced hardware and software, but they may also be introduced by other agents themselves that literally compose, compile and deploy a new agent it into the system to introduce new services[3].

In an intelligent space like Hal, is it not only desirable to introduce new agents, it is also sometimes necessary to modify running agents. This may be done in order to fix a bug, to augment an agent's services, or to provide those services in a more efficient way.

In Hal, the agents constantly evolve. As new sensory devices are introduced (e.g. more cameras, motion sensors, etc...), agents may be modified to use additional inputs. Agents may also be changed to take advantage of newly introduced agents.

The key to this goal of changing the agents in a running system is that the whole system need not come down. In a large system of diverse agents, there are independent agents that functionally have little to do with each other. Stopping agents in order to introduce or modify an agent independent of them is wasteful in terms of time and effort.

More importantly, if two programmers are working with an agent system, and one programmer needs to modify an agent, that programmer should not have to take down all of the other programmer's unrelated agents. The debugging process for a group of programmers would be too frustrating to be used in this environment!

Taking agents down may even be a waste of money. In the case of an agent-based electronic commerce system, the agent that verifies credit cards for a vendor may be broken. If the agent that shows off the goods stops along with the credit card verifier agent in order to fix only the verifier, customers can not shop, and they may go elsewhere.[4]

More important to keeping a system running is the fact that a running system of agents acquires state. When the system is taken down for modification and brought back up, the state that the running agents may have had is lost. In an asynchronous, multi-modal, real-world, interactive system like Hal, explicitly bringing the room into a desired state is sometimes quite painful, and many times a given state is practically impossible to achieve.

In debugging Hal, it may not be clear what sequence on interactions caused an observed malfunction. This makes it impossible to bring the agent system back into the state at the time of the detection of a malfunction. If the room has been used for even a few hours and a malfunction is observed, testing a bug fix could mean walking through the interactions with the room for the last couple of hours – and carrying out these interactions in exactly the same way!

---

[3] This is being done in Hal.
[4] I have personally worked on a system for controlling lottery terminals that contractually cost the vendor thousands of dollars per minute when it was down.

Not only are human interactions a part of the acquired state in Hal, but the information resources that Hal accesses change. The headlines on the CNN website may have affected the state of the room. The weather reports that Hal accesses may affect Hal's state over time.

The most critical part of the state of Hal that needs to be preserved is the information that Hal learns. Many of the systems in Hal learn through observing people in the room and interacting with them. These systems do not have any straight-forward way to unlearn, however. Trying to achieve a known state again after stopping a system of agents is made virtually impossible by the fact that many of the agents are not easily rewound.

This difficulty in achieving a particular target state which may or may not be exactly known motivates the goal of keeping as much of the system running as possible while changing some of the agents in a running system of Metaglue agents.

1.3.4 Debugging a Running Agent System

A running system of Metaglue agents should be debuggable. This is obvious isn't it? However, debugging agent systems requires non-traditional debugging strategies. Developing and debugging systems of distributed, real-time agents is a complicated matter and is a subject of research all in itself[5].

There are a number of reasons why debugging is so difficult in Metaglue. Agents in a multi-agent system are heterogeneous, i.e. each individual agent may have different

---

[5] There is M. Eng. thesis work being conducted by another Intelligent Room student on debugging running systems of Metaglue agents.

requirements and may be running in a different environment than other agents in the system. Also the problem of identifying and achieving a state of interest as described at the end of the previous section makes debugging difficult.

Agent systems may require primitives that appear in agents for explicit cooperation with the debugging system to make accessible the scope and detail of information necessary for successful debugging. There is research being done on debugging methodologies in distributed systems, and Metaglue is designed so that it can potentially leverage off of this research instead of recreating it. Metaglue has hooks in the system runtime for adding procedures that monitor the communications channels between agents.

1.4 A Brief History of Metaglue

Metaglue was developed for the Intelligent Room project and was designed closely around the perceived computational needs of Hal. Metaglue is so named because it provides primitives for expressing the computational glue that is necessary for integrating disparate agents.

There are currently several other research systems for creating software agents [ObjectSpace, IBM, GM]. They provide low-level functionality, e.g., support for mobile agents and directory services, which is necessary but not sufficient. However, they lack the low level reconnection and agent swapping features of Metaglue. Metaglue needed these features to implement mechanisms for managing groups of agents, modifying their activities in principled ways, or for abstractly and concisely describe their behaviors.

Metaglue was preceded directly by a similar agent environment and construction system called SodaBot that was also designed around intelligent environment applications. The primary focus of SodaBot was to create a programming language (SodaBotL) that would allow agent programmers to express agents in terms of human-level primitives. SodaBot included in its goals reliable networking between the agents invisible to the agent programmers and automatic distribution of agents across hosts.

However, SodaBot was too ambitious, and proved to be too difficult to learn and use. For example, variables in the code could be prefaced by over thirty different modifiers for describing their scope and persistence.

This rest of this thesis covers the design and implementation of Metaglue. The second chapter introduces the primitives of Metaglue and serves as a brief tutorial on how Metaglue agents work. The third chapter details the architecture and features of Metaglue. The fourth chapter concludes the thesis with an evaluation of Metaglue in practice, a review of some of the lessons learned in the design and implementation of Metaglue, and a discussion of future work planned for Metaglue.

CHAPTER 2

Metaglue Overview

After having established the goals of Metaglue and their motivations in Chapter 1, we now turn our attention to the actual implementation of Metaglue. We will draw our examples from Hal because the bulk of our experience is in building applications for this environment. However, we will argue that Hal is not the only useful Metaglue application. This chapter explains the fundamental agent architecture of Metaglue - how agents run, how they move, how they make use of one another, how they are maintained, and how they are configured.

This chapter introduces key Metaglue concepts, but it does not provide enough information to actually write working Metaglue agents. The examples used in this chapter are necessarily simplified in order to hide nonessential details at the introductory stage while amplifying the key architectural points. The various mechanisms and inner-workings of the system agents are detailed in chapter 3, How Metaglue Works. Unlike chapter 3, this chapter does not require a working knowledge of the Java programming language.

We will introduce Metaglue by building a simple system of agents. We begin with the simplest case – a single agent running by itself. Then we add another agent that makes use of the first and discuss how these agents are connected and distributed. Next, we examine in more detail how agents are configured and connected to one another as well has how they are maintained. Finally, we build a small multi-agent scenario. So let's get familiar with how Metaglue works!

2.1 A Single Agent

We start with a single agent that manages a VCR. The VCR agent does what one would expect: it can turn the VCR on or off and play, stop, pause, rewind, and fast-forward a videotape. Other agents such as a video-editing agent, a presentation agent, a teleconferencing agent, or an information retrieval agent can ask the VCR agent to do these things for them.

To get started, we briefly describe how a computer can control either of two types of VCR. The first type can be serially controlled via an RS-232 port and is used by broadcast professionals because of its precision control capabilities. The other, which is ubiquitous and more familiar, is accessed with an infrared remote control. This type of VCR can be controlled by a computer via an infrared emitter that sends out the same signals that a handheld remote would. We will start by writing a VCR agent that interfaces with a serially controlled VCR. In a later example in the chapter, we will write a VCR agent to interface with a remote controlled VCR through an intermediate IR (infrared) Emitter agent.

In figure 2.1, the VCR agent controls the serially controlled VCR through the serial port on Host A, where the VCR is physically connected. Notice that the VCR agent runs inside the Metaglue Virtual Machine (MVM).

The MVM provides the agent environment for all Metaglue agents. The MVM itself runs in the Java Virtual Machine that runs on the host, and thereby inherits the portability features of Java. A special *Metaglue Manager agent* locally supervises each

MVM. In this example, the Metaglue Manager agent starts the VCR agent on the MVM. But let's say the VCR agent is started on a host other than Host A. How would the VCR agent get to the MVM on Host A? We will see in the next section how Metaglue can automatically move an agent to the proper host.

2.2 The *tiedTo* Primitive

The Metaglue *tiedTo* primitive allows agents to specify a particular host where they need to run in order to function[1]. The VCR agent uses the *tiedTo* primitive in its constructor to declare that it needs to be running on Host A (with address "A.mit.edu"), the computer to which the VCR is attached. The VCR agent constructor has a line such as (This is actual Metaglue code.):

```
tiedTo("A.mit.edu");
```

If the VCR agent is started on the right host, Host A, it will stay where it is. However, if it is started on a host other than Host A, the Metaglue Manager agent on that host's MVM will arrange to have the VCR agent moved to Host A. Once it's running in the right place, the agent has access to the serial port and the VCR attached to it. Figure 2.2 demonstrates the VCR agent moving from one host to another.

Now, a whole new set of questions arise:

- What if the agent needs to move to a host where there is no MVM?

[1] In the first version of Metaglue, a particular agent may be tied to a particular host. The primitive might be extended to express needs such as a particular OS or abstract hardware resources (i.e. a frame grabber).

In this case, an MVM will automatically be started on the host via a mechanism called spreading which we will discuss in the next section. The details of spreading appear in section 3.3.2.

- Must the host name "A.mit.edu" be hard-coded in the VCR agent's *tiedTo* statement? What if the actual VCR device is later moved and reattached to another host?

We will address these questions later in this chapter in section 2.8 when we discuss agent attributes.

- How **exactly** does the VCR agent move?

This issue is addressed in section 3.1.2.2.

2.3 Spreading

If a Metaglue Manager agent is trying to move an agent to a host where there is no MVM, it will employ the spreading mechanism to start the required MVM. Spreading is the process by which a Metaglue Manager Agent establishes an MVM on a host where there was not one before so that agents can be run on that host.

Let's return to the example shown in figure 2.2 where the VCR agent moves from the MVM on Host B to the MVM on Host A. The VCR agent must be running on the

host where the VCR device is attached, but it may have been initially run on Host B. Then, the VCR agent has to move to Host A where the VCR is attached. It could also be the case that the VCR was attached to Host B when the VCR agent was started, but at a later time, the VCR is moved to Host A, and so the VCR agent needs to move with it.

Regardless of why the VCR agent needs to move, if there is no MVM on Host A, then the VCR agent cannot run on Host A. So Metaglue spreads to Host A by establishing an MVM on the target host as shown in figure 2.3. A target host must have a special daemon, called the *glue spreader*, running on it in order for Metaglue to spread to it. The Metaglue Manager agent on Host B contacts the glue spreader and has it start a MVM (along with its own Metaglue Manager agent). Then the same Metaglue Manager agent coordinates with the new Metaglue Manager agent on Host A to move the VCR agent to the newly established MVM.

2.4 Adding Another Agent

In the previous sections, we have examining a single agent with a single requirement. In this section, we will extend the example by adding another agent that uses our VCR agent.

Suppose, we have established a VCR agent on the proper host so that it can control the VCR. By itself, however, the VCR agent does nothing. Something else must ask it to do something. We will now add a Speech agent that recognizes human speech, in this case, only particular commands to the VCR, and calls on the VCR agent to carry out these commands.

Figure 2.4 shows the Speech agent running on a Windows NT host. The Speech agent relies on the VCR agent on Host A. The Speech agent has a software requirement - just like the VCR agent had the hardware requirement of being on Host A where the VCR is physically attached. In particular, the Speech agent needs to run on a machine where the speech recognition software is capable of running. In Hal, speech recognition is accomplished with the IBM ViaVoice system which only runs on the Windows NT platform.

When the user says into the microphone something to the effect of "Rewind the tape in the VCR," the Speech agent parses the sentence and directs the VCR agent to start rewinding the tape in the VCR. The actual mechanism for this speech parsing is complex, and is not described here [Coen99].

Note, the Speech agent is not running on the same host as the VCR agent. The VCR agent must run on Host A where the VCR is attached, and the Speech agent must run on a Windows NT host where the speech recognition software can run. Metaglue connects the two agents across hosts and MVMs so that the Speech agent can access the VCR agent[2]. In the next section, we will go into more detail on how Metaglue connects agents using the *reliesOn* primitive.

2.5 The *reliesOn* Primitive

The Metaglue *reliesOn* primitive connects agents so that they can request services from one another. *reliesOn* also starts needed agents if they do not already exist in a

[2] The speech interaction that has been described actually involves a multitude of agents. This example is greatly simplified.

running system of agents. We will show in this section how *reliesOn* is used to build a new VCR agent, one that provides access to a remote controlled VCR instead of a serially controlled one.

The new VCR agent requires the services of another agent, the IR Agent, that controls an infrared emitter. The IR emitter sends out remote control signals just like a handheld remote control that a person would use to control the VCR.

One of the design goals in building Hal is to try to use common, consumer electronics so that the room can be realized at a reasonable cost. In Hal, the VCR is actually an off-the-shelf, remote-controlled VCR, a common home-office component; it is not custom or computer controlled. So, the VCR agent will use the IR agent, simulating a person using a remote control.

In figure 2.5, the VCR agent is connected to the IR agent, which controls the IR emitter that sends signals to the VCR. In its constructor, the VCR agent acquires a handle to the IR agent by using the *reliesOn* primitive:

```
Agent ir = reliesOn("IR");
```

reliesOn returns a handle to an agent that can be used to issue requests to it. If the IR agent is not already running, Metaglue will automatically start it and then establish a connection. This possibility raises several questions. How does *reliesOn* know if the agent already exists somewhere, *including on other hosts*? How is the agent automatically started if it doesn't already exist? The specifics of the *reliesOn* primitive are described in detail in section 3.1.2.1.

The handle "ir" can now be used to make calls to the IR agent:

```
ir.sendSignal("PLAY");
```

The IR agent has a method "sendSignal" that makes the IR emitter attached to the local serial port send out the VCR's play signal[3].

In Metaglue, after an agent-to-agent connection is established, Metaglue will attempt to invisibly maintain that connection as long as is necessary. In the VCR agent example, the connection to the IR agent will be maintained for the VCR agent by Metaglue as long as the VCR agent is using the IR agent. Additionally, Metaglue will reconnect the two agents if one of them is stopped (e.g. for a code modification). Metaglue's mechanisms for maintaining connections is discussed later in the next section, section 2.6.

In figure 2.5, the IR agent is running on the Host A that the IR emitter is actually attached to. The IR agent is tied to Host A. The VCR agent is also running on Host A. However, the new VCR agent is no longer tied to any particular host and can run anywhere. It only relies on a connection to the IR agent, and this connection can be local or across hosts.

Some agents are both tied to some particular host and rely on one or more other agents. Notice that the Speech agent in section 2.4 was tied to a host with speech recognition software and also relied upon a connection to the VCR agent.

We have seen agents connected to other agents in order to use them. We discussed previously the fact that Metaglue should maintain the connections between agents after establishing the connections. We now introduce the mechanism in Metaglue for restoring lost connections, called reconnection.

---

[3] Each IR emitter directly attaches to a VCR's infrared receiver so that two VCRs in the same room don't

2.6 Reconnection

Let's return to the example from section 2.4 of the Speech agent that uses the VCR agent as shown in figure 2.4. By using the *reliesOn* primitive, the Speech agent gets a handle to the VCR agent on Host A with which it can issue commands to the VCR agent. Now imagine that the connection between the two agents is severed. This might be because Host A crashes, the VCR is moved and the VCR agent has to move with it, or because the VCR agent is shut down so that it can be modified and replaced. This last reason is called *agent swapping*. We will investigate agent swapping in the next section, section 2.7.

Regardless of the way in which the connection is broken, the Speech agent should not have to be restarted. Only the lost connection to the VCR agent needs to be restored. When the Speech agent receives a verbal request to rewind a tape and needs to direct the VCR agent to do so, it will discover that the connection is invalid and will attempt to reconnect to the VCR agent. Actually, it is specifically the *reliesOn* handle which discovers this problem when the Speech agent tries to use the handle. If the VCR agent is running again, even if it is running on another host, then the handle will restore its connection to the VCR agent. If the VCR agent is not running in the system, the handle will have the local Metaglue Manager agent start the VCR agent. Then the handle will find and connect to the newly started VCR agent.

We have seen in the previous sections how Metaglue establishes and maintains

---

both respond to an IR signal intended for only one of the VCRs.

connections between agents. Now we will see how agents can be removed, modified, and replaced while the rest of the system remains running.

2.7 Agent Swapping

Often, bug fixes or implementation changes are necessary for an agent running in a system of agents. As we discussed in section 1.3.3, we want to be able to modify an agent definition without taking down the whole system of agents. This is accomplished through the Metaglue mechanism called *agent swapping*.

Remember in its first implementation in section 2.1, the VCR agent controlled serial VCRs, and in section 2.4, the VCR agent was implemented to use the IR agent to control remote controlled VCRs. Imagine that we want to make this implementation switch while the rest of a running system of agents remains running. In Metaglue, the VCR agent can be removed, its definition can be modified, and the agent can be restarted using this new definition. Let's say that this was done while other agents relied on the VCR agent. The agents that rely on the VCR agent do not have to be taken down; their connections to the VCR agent are automatically restored to the modified VCR agent is restarted in the system.

Agent swapping is further discussed in section 3.4.2. Now we will turn our attention to how agents are parameterized and configured.

2.8 Attributes

We asked several questions in section 2.2 during the discussion on *tiedTo* in reference to figure 2.1: must the host name "A.mit.edu" be hard-coded in the VCR agent's *tiedTo* statement? What if the VCR is later moved and reattached to another host?

Similarly, looking at the example using the IR emitter, one might wonder: since the host for the IR emitter is currently hard-coded in a *tiedTo* statement in the constructor of the IR agent, what happens if the IR emitter device is moved from the named host to another machine? Clearly, the IR agent would be rendered useless. But agents are not actually written with hard-coded arguments as in the examples. They use dynamic *agent attributes*.

When an agent declares an attribute, the agent gets a handle into a Metaglue database called the Attribute Manager that holds site-wide and dynamically changeable configuration parameters (see section 3.2.4 for more about the Attribute Manager). For example, the IR agent can define an attribute called "location":

```
Attribute location = new Attribute("LOCATION");
```

This attribute must then be defined for the IR agent in the attribute database. The value of "LOCATION" is the host to which the IR emitter is attached at any given time. Now the *tiedTo* call can use the handle to the location attribute:

```
tiedTo(location.getValue());
```

An agent may have any number of the globally managed attributes used for storing any dynamic information for the agent. In this example, the value of the attribute named "location" defined for the IR agent provides a dynamic value for the *tiedTo* call so

that the agent is properly situated on the host where the IR emitter is attached. When the IR emitter is moved, value for the attribute is changed in the attribute database. Agent attributes can be used to hold any configuration parameters, not just locations for *tiedTo* statements. The attribute database is kept up to date as configuration parameters change for the agents. The Configurator agent is used to maintain the attributes in the database (see figure 3.1 and section 3.2.4).

2.9 An Extended Example

We have seen some very simple examples of how Metaglue agents are automatically distributed and connected. Now let's catch a glimpse of what is involved in a larger system of agents in order to appreciate how quickly complexity sets in.

Figure 2.6 shows the agents used in the couch scenario in Hal described in chapter 1. The agents are distributed and connected according to their particular needs. The IR agent is tied to the host where the IR emitter is attached, the X-10 agent is tied to the host where the X-10 signaler is (X-10 controls the power to devices), and the "lie" detector, the agent which detects from video images whether someone is lying down or not, is tied to a host with the frame grabber (connected to a camera pointed at the couch).

Some agents involved in the scenario are also tied to hosts where certain software can run. The speech recognition software used in Hal only runs on a Windows platform, and so the agent for speech recognition must be tied to a host running Windows.

The Nap agent located in the center of the figure is the agent that coordinates the detection of somebody lying down for a nap and it controls the room's environment,

making the room a suitable space for napping as necessary. The Nap agent also coordinates the interaction for setting a wakeup alarm. Notice that there is a chain of agent reliance from the Nap agent to every other agent in the diagram.

By simply starting the Nap agent in Metaglue, all of the agents required for the scenario are automatically started and distributed to the appropriate hosts that fulfill their individual needs. Imagine the routine for establishing even this simple scenario manually – remembering which agents go where, getting them connected in the right order, and remembering their configuration parameters. The situation is made more difficult when the agents and the environment are changing.

Let's say we establish an MVM on a host and then start the Nap agent on that MVM. In figure 2.6, we see that the Nap agent directly relies on the Alarm, CD, Lighting, and "Lie" Detector agents. Metaglue will start those agents when the Nap agent is started. In figure 2.6, the Alarm, CD, and Lighting agent all start on the same host as the Nap agent that relies on them because they are not tied to any particular host. However, the "Lie" Detector agent requires that it run on a host with a frame grabber for vision processing. This requirement is detected during the startup of the "Lie" Detector agent, so Metaglue moves it from the default startup host with the Nap agent to the proper host.

If the host with the frame grabber does not have an MVM running on it, Metaglue will "spread" to this host. The Metaglue Manager agent supervising the Nap agent will call on a special daemon running on the target host with the frame grabber to start an MVM. Then the "Lie" Detector agent can be run on the newly started remote MVM. The arrows in figure 2.6 from the Nap agent to the four agents that it relies on represent

the communication channels between the Nap agent and those other agents. The direction shows that the Nap agent requires the services of these agents and not the other way around. For example, if the Alarm agent was started in isolation, the Nap agent would not be started because the Alarm agent does not rely on the Nap agent.

The "Lie" Detector itself relies on the Mux agent, the agent which controls an audio/visual multiplexer. The Mux agent will be started and then appropriately moved to the host which is connected to the A/V multiplexer as shown in figure 2.6. There are many cameras and frame grabbers for different hosts connected to the A/V multiplexer in Hal, but only the two we are interested in are shown in the figure.

The "Lie" Detector agent, after getting the handle it needs to the Mux agent, will immediately ask the Mux agent to connect the couch camera to the frame grabber on the host where the "Lie" Detector agent resides. Now the "Lie" Detector agent begins to continuously scan the in-coming image for hints of somebody laying down on the couch. The Nap agent will use its handle to the "Lie" Detector agent to register with the "Lie" Detector agent for "lying-down" events. The Alarm, CD, and Lighting agents and the agents they rely on are similarly started and appropriately distributed so that when everything is started, the system is laid out as shown in figure 2.6. Only the Nap agent was manually started.

Now that the system of agents is established, let's look at how this system would respond to an occupant lying down on the couch in the room. First, the person lying down would appear in the image from the camera over the couch. The signal from the couch camera is multiplexed to the frame grabber in the host where the "Lie" Detector is running. The "Lie" Detector uses machine vision techniques to scan the images coming

into the frame grabber and identifies the person lying down on the couch.
Since the Nap agent has registered with the "Lie" Detector agent for notification when a person lies down on the couch, the "Lie" Detector agent will post this event to the Nap agent. The Nap agent may use other data as evidence that the occupant is taking a nap in addition to the notification that somebody is lying down, but in this simple example, we will just equate lying down with napping.

After determining that the occupant is going to take a nap, the Nap agent will adjust the room's environment so that it is conducive to napping according to the occupant's preferences. It will have the Alarm agent set a wake-up time for the occupant. The Nap agent will also register for the wake-up event. In order to set the wake-up time, the Alarm agent uses the Speech Synthesizer agent to ask the person what time they want to wake up. When the person answers, the Speech Recognizer agent will parse the utterance and send back the extracted time to the Alarm agent.

The Nap agent will also tell the CD agent to put on some soft music according to the occupant's preferences. Since the CD player in Hal is remote-controlled, the CD player uses the IR agent to send a signal to the CD player to start playing a particular series of songs.

Finally, the Nap agent directs the Lighting agent to decrease the amount of light in the room. The Lighting agent itself relies on the services of the Drapes, Blinds, and Lamp agents to accomplish this. It tells the Drapes and Blinds agents to close the drapes and blinds. It also tells the lamp agent to dim or turn off the halogen lamps in the room. Although not shown in this example, the Nap agent may do a great deal more such as telling the VCR agent to tape a show that the sleeping occupant habitually watches, it

may direct the Telephone agent to hold calls, etc… When the "Lie" Detector agent detects that the occupant is no longer lying down, it will inform the Nap agent which will then instigate a "waking-up" routine.

Figure 2.7 provides a glimpse at the current complexity of Hal, where over eighty software agents interact to form a futuristic command post. Although we don't explore this system here, it illustrates the complexity permitted by Metaglue.

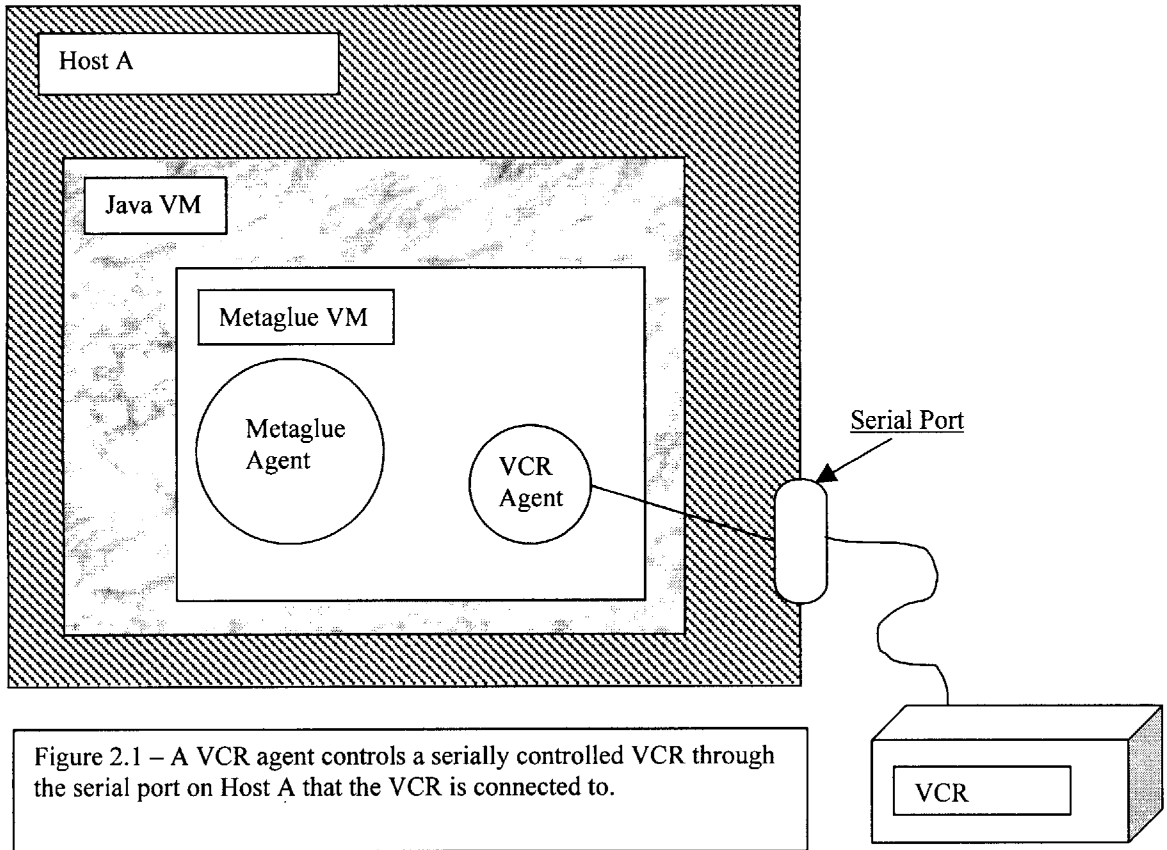


Figure 2.1 – A VCR agent controls a serially controlled VCR through the serial port on Host A that the VCR is connected to.

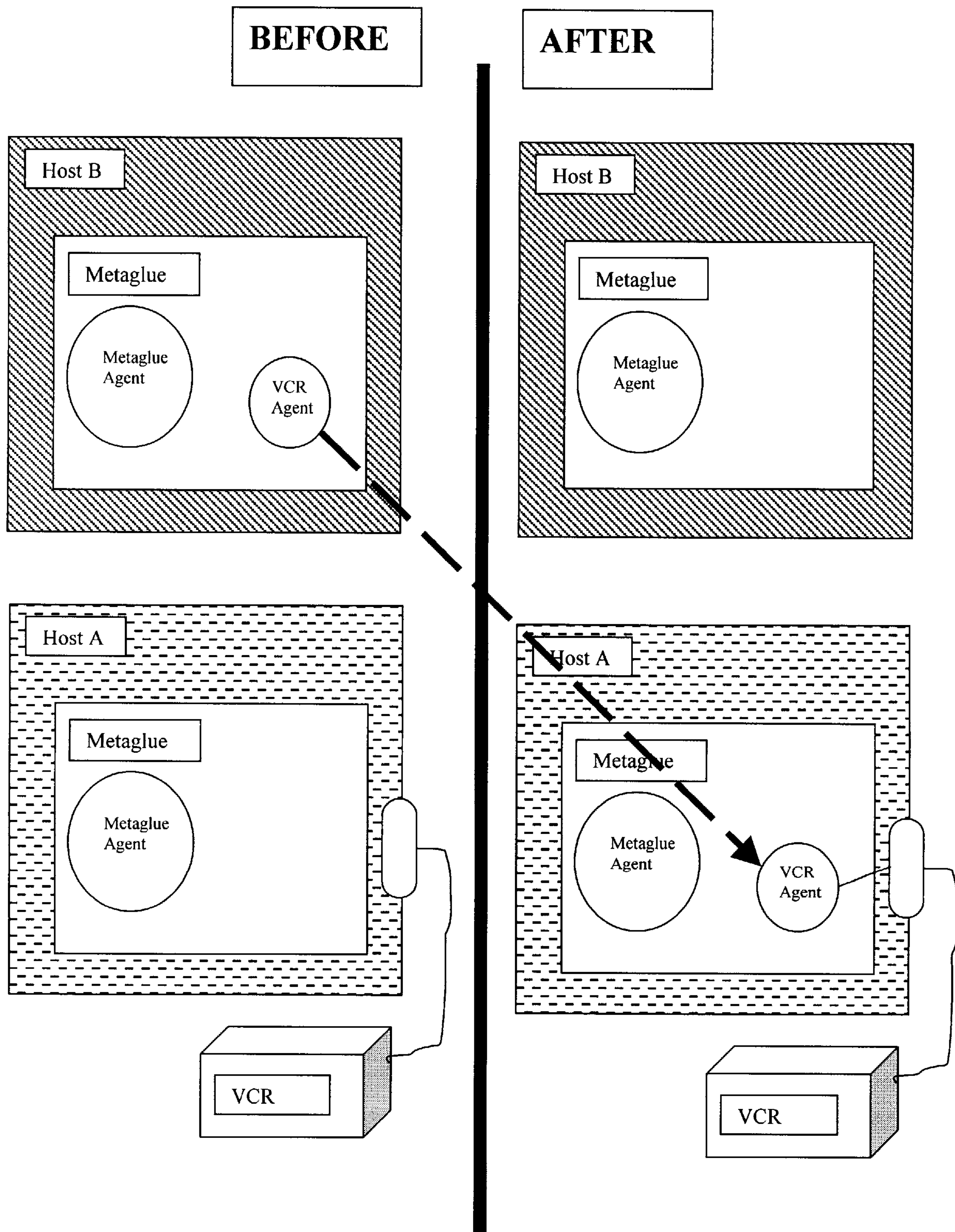


Figure 2.2 – The *tiedTo* primitive used in the VCR agent moves the VCR agent from Host B to Host A where the serially controlled VCR is attached.

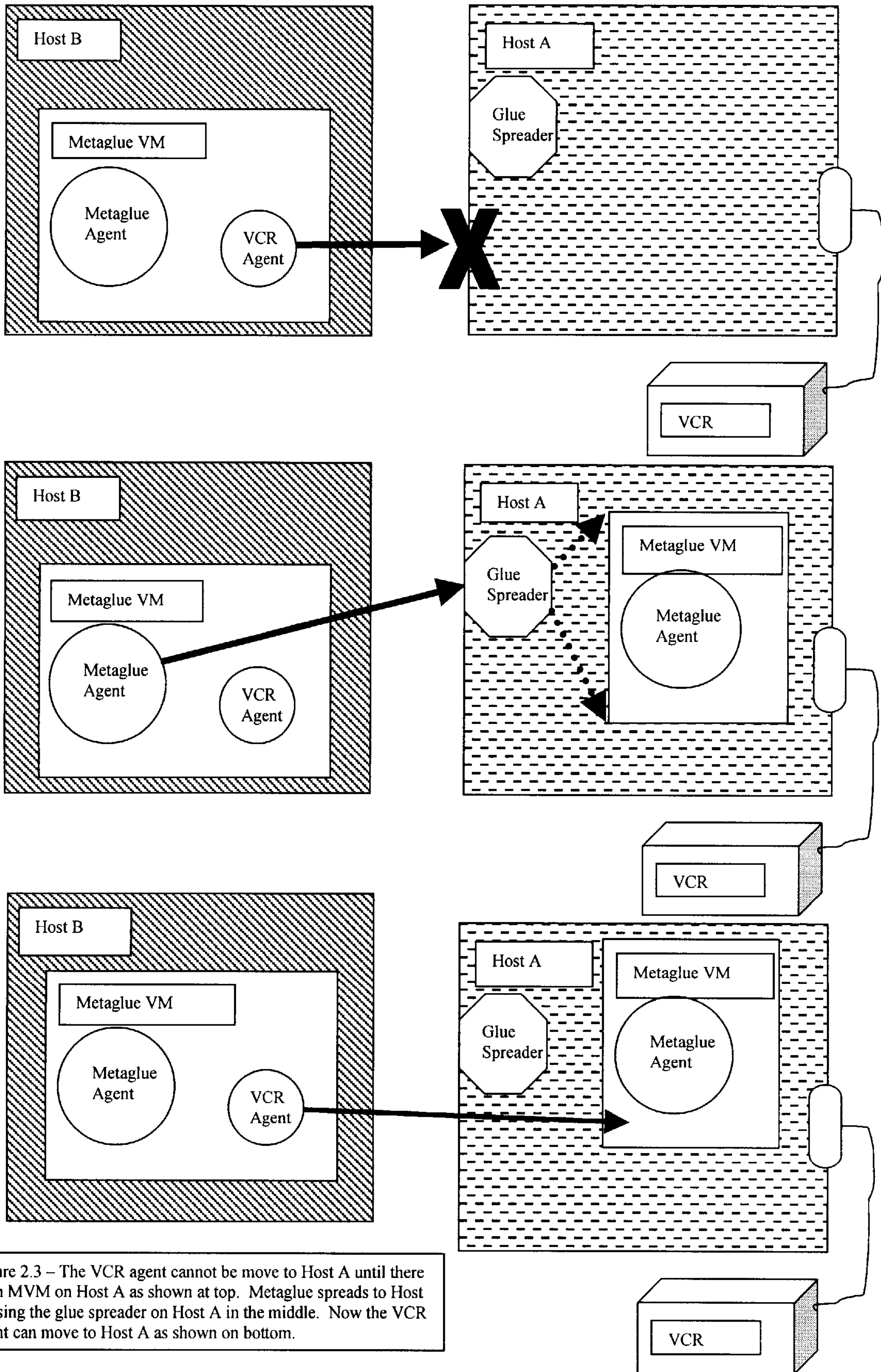


Figure 2.3 – The VCR agent cannot be move to Host A until there is an MVM on Host A as shown at top. Metaglue spreads to Host A using the glue spreader on Host A in the middle. Now the VCR agent can move to Host A as shown on bottom.

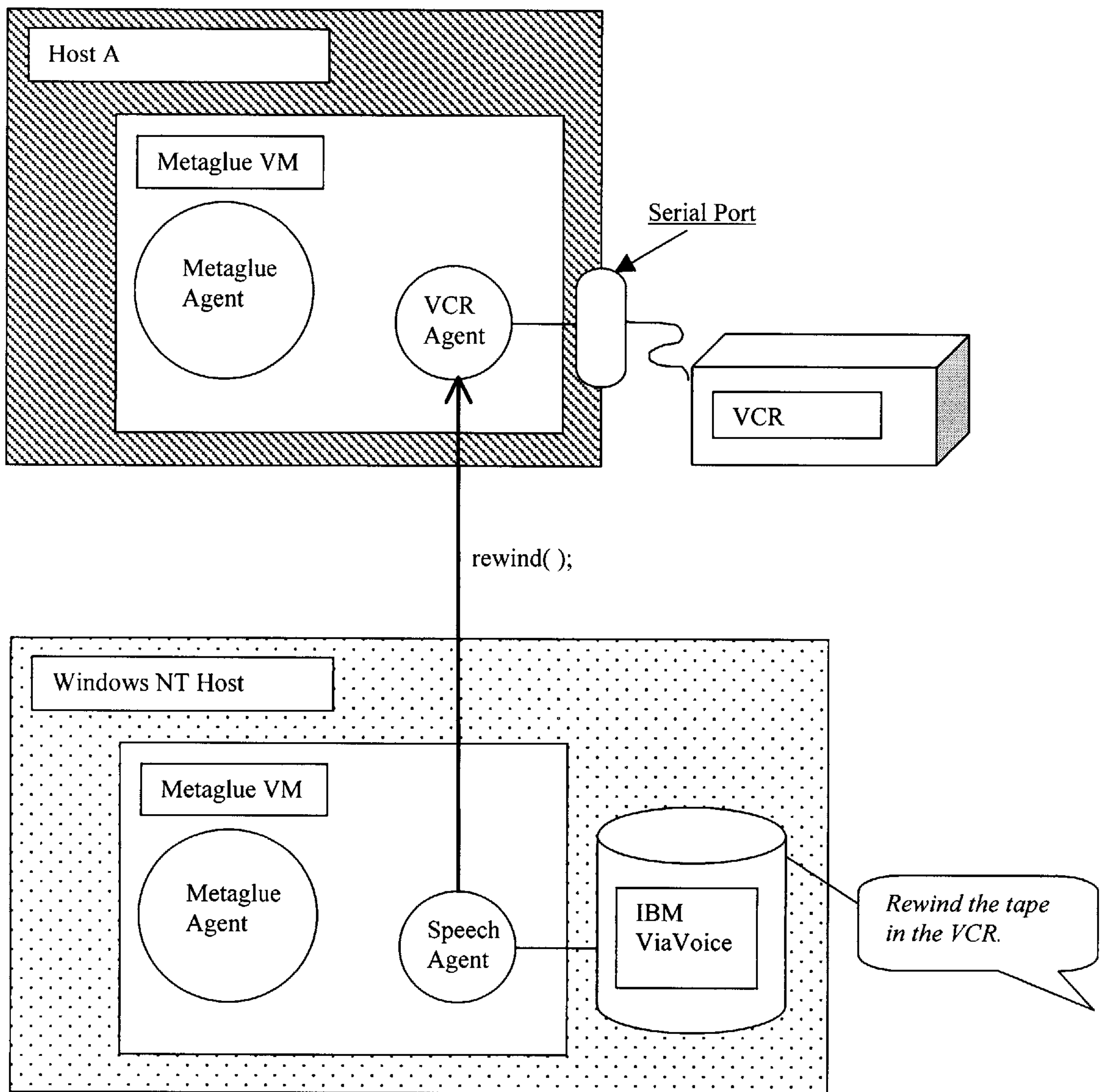


Figure 2.4 – The Speech agent on the Windows NT host with the IBM ViaVoice speech recognition software receives input from ViaVoice and directs the VCR agent on Host A where the VCR is attached.

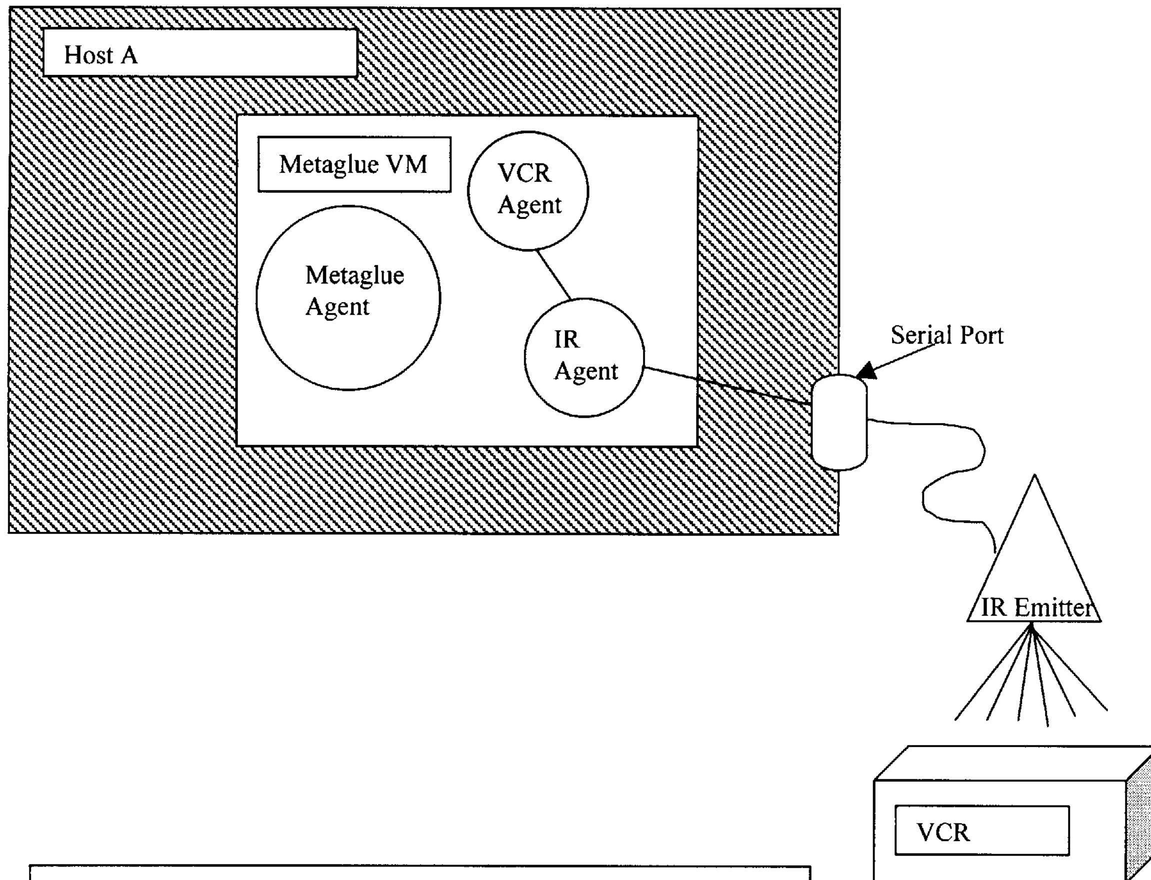


Figure 2.5 – A VCR agent uses the IR agent that controls an IR emitter via a serial port to control an infrared remote VCR.

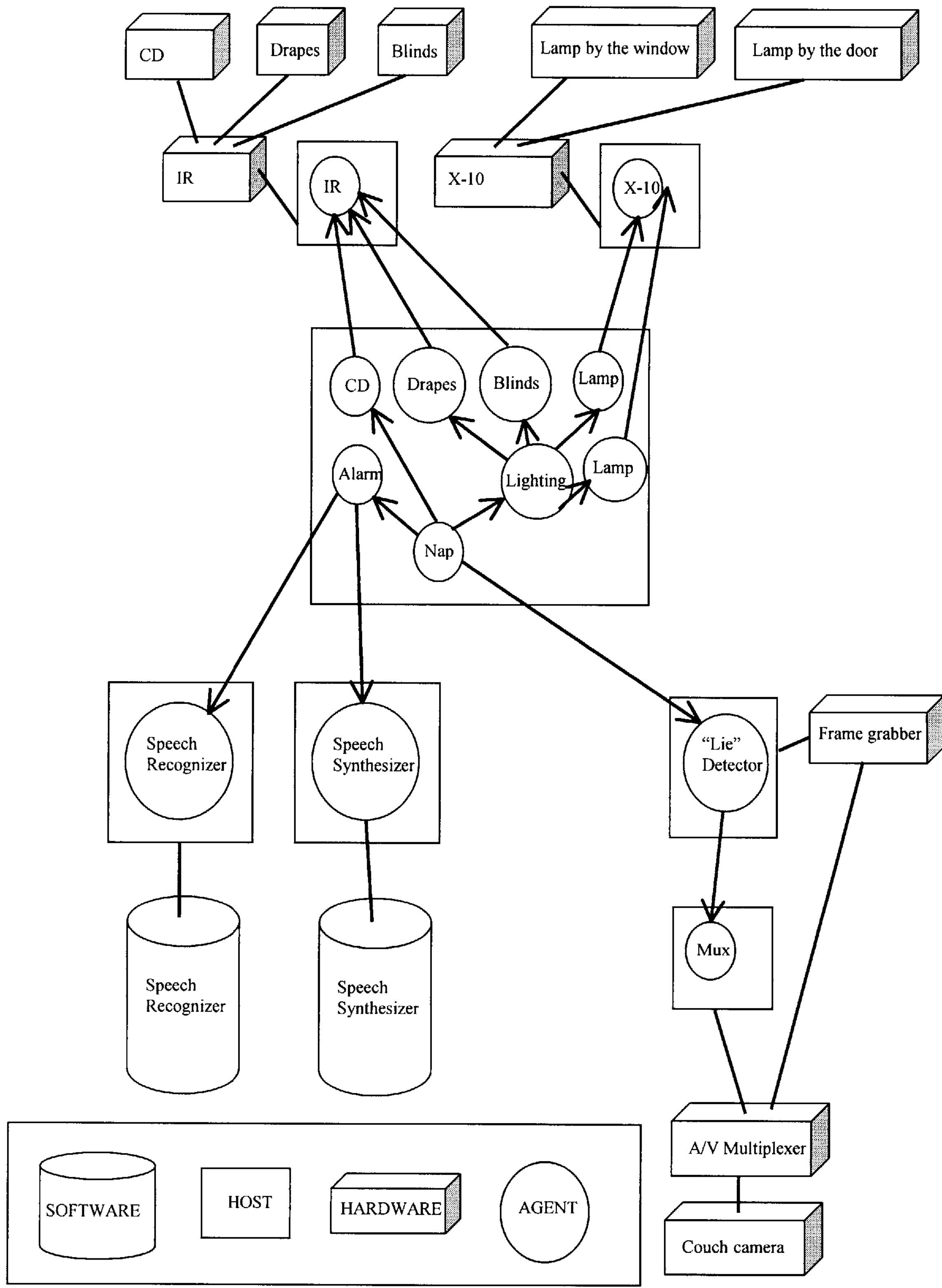


Figure 2.6 – The couch scenario is coordinated by the Nap agent in the middle. . The arrows point in the direction of reliance between agents (but not hardware or software components).

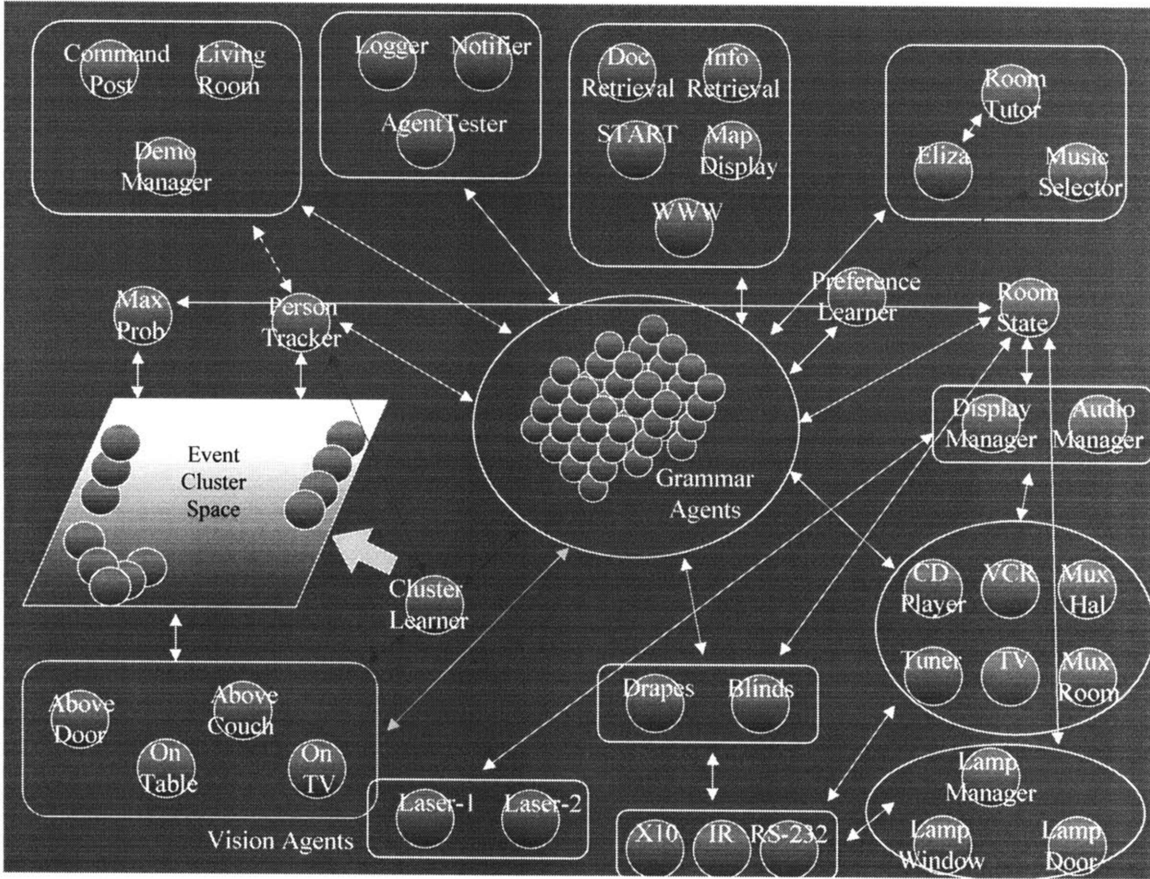


Figure 2.7 – A glimpse of the command post agent architecture in Hal.

CHAPTER 3

How Metaglue Works

The first chapter set the scene for Metaglue and motivated the Metaglue system. The second chapter introduced the key primitives and mechanisms of Metaglue with examples. This third chapter will elucidate the details of the Metaglue system. In this chapter, we assume you are familiar with Java and the general architecture of Metaglue from chapter 2.

First, we will describe Metaglue agents. We will also discuss some special agents, the system agents. Then we will discuss the Metaglue Virtual Machine and spreading, the distribution of Metaglue agents among hosts. Finally, we will discuss the two Metaglue mechanisms for keeping a system of agents up and running, reconnection, and dynamic swapping, followed by a look at a few simple debugging aids.

3.1 Metaglue Agents

We have been using the term "agent" interchangeably both to mean the definition of a type of agent (as in "the VCR agent") and to mean a specific instantiation of an agent definition (as in "a running system of agents"). Now we distinguish the two meanings and discuss how instantiated agents are identified. Following that, we more formally describe agent definitions.

3.1.1 Agent Naming Scheme: AgentIDs

An agent definition is named by its occupation, namely the set of skills that the instantiated agent is intended to provide. A particular instantiation of an agent is identified by an *AgentID*. An AgentID is made up of three fields: a society, the occupation of the instantiated agent, and a designation.

3.1.1.1 Societies

Instantiated Metaglue agents are grouped into societies[1]. Within a society, there may be agents that do not rely on each other, directly or indirectly, that is, there may be islands of intercommunicating agents. Agent societies do not have to be completely connected. Agents in a society, by default, tend to search for the agents they need in their own society.

Agent societies are used in three major ways. The first is that they are used to name a group of agents working together on a local collection of hosts. Essentially, a system of agents is local to a site that might be behind a firewall, such as the society of agents in the MIT AI Lab. Special agents may then be built to find agents at other sites and introduce their services to a given site that is lacking the services.

The second use of agent societies is that even in a particular site, individuals may wish to have their own isolated collection of agents for some purpose. In particular, two individuals may want parallel collections of agents for debugging and experimentation purposes.

[1] The grouping of agents into societies is influenced by Marvin Minsky's *The Society of Mind* [Minsky]. Currently in Metaglue, societies serve as a naming and scoping mechanism.

A third way that Agent societies are used is to group agents in a local site into a collection of smaller sites. For example, each room in a house might have its own society of agents. Societies provide for the nominal grouping of agents whether that name is a site name such as “ai.mit.edu”, an individual like “luke”, or a sub-site name such as “kitchen”. The society field in the AgentID identifies the unique society that an instantiated agent belongs to.

3.1.1.2 Occupations

An instantiated Metaglue agent provides a set of methods, called its *interface*, that is identified by the occupational name of the agent definition from which the agent was instantiated. Typically, the definition’s occupational name is the same as the interface which it defines (plus an “Agent” suffix; the occupation name used in the AgentID does not include the “Agent” suffix). For example, the occupation field in the AgentID of an instantiated lamp agent with a definition named “LampAgent” that implements the “Lamp” interface is “Lamp”. The occupation field in the AgentID for an instantiated agent identifies what definition the agent was instantiated from and essentially names its skills.

3.1.1.3 Designations

A particular society might contain multiple instantiated agents of the same occupation, and these need to be uniquely identified. In the example shown in figure 2.6,

there are two lamp agents, one for the lamp by the door and one for the lamp by the window. They are in the same society and they are both lamp agents providing the same methods so in their respective AgentIDs they have the same society and occupation fields. Their designations, however, are different. One has the designation "door" and the other has the designation "window".

One important agent programming rule-of-thumb when using designations is that they should **not** be used to carry dynamic information that would be better kept in the Attribute Manager (see section 3.1.2.3). They should only serve to distinguish agents. Agent specific information should come from the Attribute Manager.

3.1.1.4 Nomenclature

An AgentID is written SOCIETY:OCCUPATION-DESIGNATION. The AgentID of the instantiated agent controlling the lamp by the door in the "hal" society is written "hal:Lamp-door". Every AgentID has a society, occupation, and designation, but the society and designation are possibly empty strings. However, an empty society or designation remains a part of the agent's identification and does not indicate that the agent does not belong to a society or have a designation. It is not uncommon for the designation to be the empty string, as many times it only makes sense to have one of a particular type of agent in a society (e.g. agents that coordinate scenarios). When the designation is the empty string, the "-" is not included in the written AgentID; the ":" is included whether the society is the empty string or not. The occupation should not include ":" or "-" characters, and it is never the empty string.

3.1.2 Agent Definitions

An agent definition has four main parts - a name, its parent, its interface (the list of methods the agent provides) and the actual implementation of the methods of the agent. As was previously discussed, the name of the definition is the occupation of the agent (with the "Agent" suffix added).

The parent is another agent definition that this agent inherits methods from. All agents ultimately, directly or indirectly, inherit from the root ancestor the *Agent agent.* A particular implementation mandates whether an agent can directly extend more than one parent or not. The implementation of Metaglue that accompanies this thesis is single inheritance because Java is.

The interface may be implicit, for example, using tags to identify in the definitions which methods are exported agent skills and which are internal support procedures for the agent. On the other hand, the interface may be an explicit list of exported agent skills as it is in the implementation of Metaglue that accompanies this thesis using Java interfaces.

The definitions of the methods comprise the core of an agent definition. These methods implement the services that an instantiation provides. Metaglue defines a number of primitives that agent programmers can use in writing these definitions. We will discuss them in the sections that follow.

3.1.2.1 The *reliesOn* Primitive

The *reliesOn* primitive is used to tell the Metaglue system that an agent requires a connection to another particular agent. A successful call to *reliesOn* will result in a handle to the required agent. The required agent may already exist, or if it doesn't already exist, Metaglue automatically starts it.

The returned handle is used by the agent to call the methods of the other agent. This handle does not have to be maintained by the agent in terms of connectivity. If the required agent moves, crashes, or the connection is otherwise broken, Metaglue will restore the agent or find and reconnect to the agent as necessary.

There are a number of similar forms of the *reliesOn* primitive:

```
Agent reliesOn(AGENTID)
Agent reliesOn(OCCUPATION, DESIGNATION)
Agent reliesOn(OCCUPATION)
```

The first form identifies the required agent using the full AgentID, the society, occupation, and designation for the required agent. Recall the *reliesOn* used in the example VCR agent in the second chapter[2]:

```
        Agent ir = (IR) reliesOn(new AgentID(getSociety(),
"agentland.output.IR", "A"));
```

In this example, we build a full AgentID using the Metaglue primitive *getSociety* that returns the society that the calling agent is a part of (see section 3.1.2.4). Most agents rely on other agents in their own society, however, and so the second form of

[2] Agent definitions are stored according to the Java packaging scheme. The IR agent definition is in the "agentland.output" package. Also, there are a number of IR emitters in Hal, each programmed to broadcast different signals. The IR emitter designated "A" broadcasts VCR infrared signals.

reliesOn is used where the society is assumed to be the society of the calling agent:

```
Agent ir = (IR) reliesOn("agentland.output.IR", "A");
```

For some agents, there is commonly only one such agent per society, and so the designation is the empty string. For example, scenario coordinating agents, such as the Nap agent from chapter 2, only appear once in a society, and so are designated as the empty string. Some other agent that uses the Nap agent would rely on the nap agent using the third form of *reliesOn* where the designation is implicitly the empty string:

```
Agent ir = (IR) reliesOn("agentland.scenario.Nap");
```

The *reliesOn* primitive contacts the local Metaglue Manager agent supervising the MVM that the calling agent is running on and requests that it produce the desired agent. The Metaglue Manager agent then contacts the site's *Catalog agent*, which knows where all agents are running, to find the desired agent. If the agent is listed, the Catalog agent will verify that the agent is alive and accessible and will return a handle back to the Metaglue Manager agent who in turn hands it back to the calling agent.

If the agent is not listed or it is found dead, then the Catalog agent reports this to the Metaglue Manager agent. In this case, the Metaglue Manager agent will start the agent locally itself. Since the startup procedure results in the started agent being registered with the Catalog agent, the Metaglue Manager agent can then check with the Catalog agent again, the agent will be there, and the handle is returned to the calling agent.

There are two ways that the system protects against timing errors in this

mechanism. First, the methods of the Catalog agent (i.e. registering and deleting agents) are atomic. The list of agents at a site is also synchronized since there is one Catalog agent for a site that maintains this list.

Second, the returned handles are persistent, so if the agent that the handle refers to moves or dies, the handle will automatically restore a connection to the agent or have the agent restarted as necessary. This mechanism is called reconnection (see section 3.4.1). Say, for example, that the Catalog agent verifies that an agent is alive and accessible in a system and returns a handle, but the agent subsequently shuts down. When the handle is used, the call to the agent will fail, but the system will restart the agent, the handle will be fixed, and the call will be retried.

Notice that when the Metaglue Manager agent has to start the desired agent it does not immediately return the handle but instead goes again to the Catalog agent for the handle. This is because first, even though the local Metaglue started the process of starting the agent, it may not finish this process. Agents can move (via *tiedTo*) to another host, where the Metaglue Manager agent supervising the remote MVM finishes the job. So the local Metaglue Manager agent may not even have direct access to the handle.

Second, going to the Catalog agent provides a single synchronization point. If two Metaglue Manager agents simultaneously are starting the same agent, the first one done will register, and the Catalog agent will not allow the second one to register, so the agent will die. The Metaglue Manager agent that started the second one will go to the Catalog agent for the handle to the successful agent.

Finally, in order to implement the reconnection and reloading features of Metaglue, the handles need to be massaged and registered as connections. This is most

easily done in one place in the system, in the Catalog agent.

3.1.2.2 The *tiedTo* Primitive

The *tiedTo* primitive is used to tell the Metaglue system that an agent needs to be running on a particular host. This could be because the agent needs access to hardware or software found only on the named host. It could also be because there is another agent on that host which this agent communicates heavily with and being on the same host would be more efficient[3].

If the agent is already running on the correct host, then the agent does not move. Otherwise, the agent is stopped and restarted on the proper host. If there is no MVM on the destination host, Metaglue starts one via a process called "spreading" (see section 3.3.2).

The syntax of the *tiedTo* primitive is as follows:

```
tiedTo(HOST);
```

The HOST designates the destination host of the agent and is an IP address or a host by name. An example was seen in the example IR agent from chapter 2 that ties the IR agent to the host where the IR beacon is physically attached, Host A:

```
tiedTo("A.mit.edu");
```

The host should not be hard-coded, however. The IR agent should have a

[3] A connection between agents on the same host is a faster than one between two different hosts.

dynamic, global attribute which defined which host the appropriate IR beacon was attached to. A handle to the attribute would be used as the HOST argument to *tiedTo*. The next section describes attributes in Metaglue.

The *tiedTo* primitive first checks to see if the agent needs to move. If it doesn't (it's on the correct host) then *tiedTo* done. Otherwise, it contacts the local Metaglue Manager agent supervising the MVM and requests that it find the Metaglue Manager agent supervising the MVM on the destination host. The Metaglue Manager agent asks the Catalog agent. If the remote Metaglue Manager agent is found, a handle is returned to the Metaglue Manager agent which in turn returns it to *tiedTo*. The *tiedTo* primitive then instructs the Metaglue Manager agent supervising the destination MVM to restart the agent and it shuts down the local agent.

If, on the other hand, the destination MVM does not exist on the desired host, then *tiedTo* initiates spreading to the host which starts up an MVM on the remote host along with its supervising Metaglue Manager agent. The remote Metaglue Manager agent will register with the Catalog agent, and so when the *tiedTo* primitive has the local Metaglue Manager agent check with the Catalog agent again, *tiedTo* will get the handle to the remote Metaglue Manager agent and complete the move of the agent as previously described.

3.1.2.3 The Attribute Primitives

Attributes are adjustable parameters associated with each agent. These parameters are dynamic, that is, they can be adjusted in real time by accessing the

Attribute Manager agent through the Configurator Agent (see figure 3.1). The value of a particular attribute can be set specifically down to the granularity of the complete AgentID of an instantiated agent. So, for example, the "port" parameter in the IR agent would have a different value for the IR agent with AgentID "brent:IR-A" controlling IR beacon A than for the IR agent "brent:IR-B" controlling IR beacon B when the two IR beacons are attached to the same host. IR beacon A may be attached to port "/dev/ttyrd" and IR beacon B may be attached to "/dev/ttyrg".

If there is no set value for a parameter for a specific AgentID, then Metaglue looks for the value of the empty string designation under the same society and occupation (the empty string designation is often considered to refer to a default agent). If that too is lacking, then an error is generated, and the value must be set before the agent can continue.

An agent declares its attributes. The values for the attributes must also be set in the central, site attribute database. Forgetting to set a value for an attribute is a common source of error. We declare an attribute in the IR agent:

```
Attribute port = new Attribute("PORT");
```

Here, a new attribute is declared with the name "port" and assigned to a handle to the attribute. When the agent needs the value of port attribute, it uses the handle:

```
    VALUE port.getValue();
```

This call returns the value of the port attribute (i.e. "/dev/ttyx") according to the full AgentID of the calling instantiated agent which can be used by the agent to communicate with the correct port.

3.1.2.4 The Naming Primitives

The naming primitives provide an agent with the parts of its own AgentID:

```
SOCIETY getSociety()
OCCUPATION getOccupation()
DESIGNATION getDesignation()
```

This information is available even in the constructor of the agent (which is where it is typically used).

3.1.2.5 The *shutdown* Primitive

The *shutdown* primitive causes an agent to close its connections and die. The agent does not provide service after it is dead. The *shutdown* primitive is not actually meant to be called, however; it is meant to be overridden. *shutdown* is analogous to the *destroy* method in C++. An agent programmer defines a shutdown procedure specific to the agent which may clean up any communications channels that it explicitly opened (not the connections that Metaglue maintains), commit some work, close any processes it started, etc. A sample procedure is defined as such:

```
public void shutdown() throws RemoteException {
        COMMIT WORK
        CLEAN UP
}
```

The overriding shutdown procedure should not be explicitly called for a proper killing of an agent. Only the Metaglue system calls this method. Instead, in order to request that an agent be killed, the *shutdownAgent* method of the Metaglue Manager agent supervising the local MVM should be used because it does some system level cleanup work including informing the Catalog agent. The Metaglue Manager agent's *shutdownAgent* method internally calls the agent-specific *shutdown*. The primitive is only meant to allow agent programmers to define specialized shutdown tasks for agents in addition to the shutdown tasks that apply to all agents.

3.2 Metaglue System Agents

We have discussed the naming, definitions, and primitives of Metaglue agents in general. Now we turn our attention to some actual agents. These agents are Metaglue's system agents, the agents that are a necessary part of any running system of agents.

3.2.1 The Agent Agent

The Agent agent is the system agent that all agents ultimately extend from either directly or indirectly. This includes even the other system agents like the Metaglue Manager agent and the Catalog agent. In this sense, Metaglue is meta-linguistic – the Metaglue system itself is built out of agents. The Agent agent provides the preliminary work that must be done during the startup process of all agents, namely it prepares the agent for connectivity to other agents, establishes a connection with the local Metaglue

Manager agent, and it acquires the agent's naming information. More importantly, the Agent agent provides all agents with the Metaglue primitives described in the previous sections; the Metaglue primitives comprise the Agent agent's interface. An agent is not a Metaglue agent if it is not a descendent of the Agent agent.

3.2.2 The Metaglue Manager Agent

The Metaglue Manager agent is the system agent that supervises an MVM. The Metaglue Manager agent searches for agents in a system of agents and starts agents and shuts them down on the MVM that it is maintaining. Now we will describe the methods of the Metaglue Manager agent.

3.2.2.1 The *findAgent* Method

The *findAgent* method has the following form:

```
Agent findAgent(AGENTID);
```

findAgent checks with the Catalog agent to see if the agent in question is alive and in the system of agents. If so, then *findAgent* returns a handle to the caller, otherwise it generates an error signaling that there is no such living agent in the system. *findAgent*'s primary use is by the *reliesOn* primitive to see if the agent is already in the agent system.

3.2.2.2 The *startAgent* Method

The *startAgent* method has the following form:

```
startAgent(AGENTID);
```

startAgent loads in the agent definition according to the occupation in the AgentID. It then instantiates a new agent by running its constructor and filling in some information (its name, for example). Finally, *startAgent* registers the new agent with the Catalog agent.

If the agent cannot start or the Catalog agent reports that there is already such an agent, then *startAgent* generates an appropriate error. The primary use of *startAgent* is by the *reliesOn* primitive to start up an agent that is not already in the agent system.

3.2.2.3 The *shutdownAgent* Method

The *shutdownAgent* method has the following form:

```
shutdownAgent(AGENTID);
```

shutdownAgent asks the Catalog agent to remove the agent identified by the AgentID from the Catalog. It then has the agent close its connections to other agents. Finally, it invokes the shutdown procedure specific to the agent and leaves the agent disconnected from the agent system. The agent is now dead and will be cleaned up by the garbage collector. *shutdownAgent* is the method that should be used to properly remove an agent from an agent system and not the shutdown primitive of the agent (*shutdownAgent* calls this appropriately).

3.2.3 The Catalog Agent

The Catalog agent is the system agent that keeps track of the locations of agents in a running system of agents as well as their interconnections. It is the definitive source of exactly which agents are in a running system of agents, and it is the official system distributor of handles to agents. All instantiated Metaglue agents are registered with the Catalog agent. It serves as the synchronization point for ensuring name uniqueness among the agents and guarantees continuity across the system. It will reject an attempt to add an agent with the same AgentID, and it is the only source for connections to the agents.

In the current implementation of Metaglue, if the Catalog agent crashes, all of the existing agents become inaccessible. In order to protect against this, the Catalog agent should keep a disk backup of the list of agents in the site so that when the Catalog is restarted, the state can be restored.

The Catalog agent has the ability to add, delete, and lookup agents. Adding and deleting are straightforward. We will examine the Catalog agent's ability to find an agent in the system and provide a connection to it.

3.2.3.1 The *lookup* Method

The *lookup* method is primarily used by the *tiedTo* primitive to find Metaglue agents on various hosts and MVMs. It is also used by the Metaglue Manager agents to

find existing agents in the system for the agents on their platforms. The *lookup* method takes the following form:

```
Agent lookup(AGENTID);
```

lookup does three main things for every query it services. First, it checks in the catalog of registered agents in the system to see if an agent with a matching AgentID exists. If it does not, *lookup* generates an error. If it does exist, *lookup* then tests to see if the agent is actually alive; if the agent crashed, it may not have properly unregistered. If the agent is dead, it is unregistered and reported as non-existent. If the agent is alive, *lookup* prepares a new reliable connection to the agent and hands this handle back to the caller (see section 3.4 for more about reliable connections).

3.2.3.2 The Catalog Monitor Agent

The Catalog Monitor agent is the agent that presents a graphical map of the running agents in a system of agents (see figure 3.2). It registers with the Catalog agent for catalog events. When the Catalog agent adds an agent, it tells the Catalog Monitor agent. When the Catalog agent removes an agent, it tells the Catalog Monitor. Indeed, any agent can register for these events with the Catalog agent.

When the agent icons presented by the Catalog Monitor agent are clicked on, information about the agent including its methods are shown. The methods can actually be executed from here for direct control of the agent. The Catalog Monitor agent serves as a simple debugging tool for a system of agents in Metaglue.

3.2.4 The Attribute Manager Agent and the Configurator Agent

The Attribute Manager agent maintains the site's attribute database. Every unique AgentID can have an individual value for each attribute in this database. The attribute primitives use the Attribute Manager agent to get values for the attributes that specific agents need.

The Attribute Manager agent can also be used to add attributes and change attribute values. This has the effect of centralizing and synchronizing global configuration parameters for the agents. The Configurator agent, like the Catalog Monitor agent, is a graphical tool for browsing the attribute database and changing values (see figure 3.1). Like the Catalog Monitor agent, the Configurator agent registers with the Attribute Manager agent for attribute events and reflects changes to the database when it receives those events.

3.3 The Metaglue Virtual Machine and Spreading

The Metaglue Virtual Machine (MVM) provides the virtual machine for interpreting and running Metaglue agents. MVMs are automatically started as needed on hosts using glue spreaders.

3.3.1 The MVM

Metaglue agents run on MVMs which in turn run in Java Virtual Machines. The MVM provides the execution environment for the agents. MVMs are supervised and maintained by Metaglue Manager agents. The MVM is light-weight in terms of code and is a part of the Metaglue Manager agent definition. An MVM can be invoked from a command line.

The MVM is defined as a special method of the Metaglue Manager agent and is static, meaning that it does not require an instantiation of the Metaglue Manager agent. The first thing that the MVM does is instantiate a Metaglue Manager agent to supervise the agents that will run on the platform. It then establishes some security restrictions for the agents running on the platform (agents should not be able to indiscriminately kill the MVM, for example). It will also start a Catalog agent and Attribute Manager agent if necessary. Finally, the MVM tells the local Metaglue Manager agent to start up any agents given at the command line.

3.3.2 The Glue Spreader

Starting an MVM on a host where one is needed is called *spreading*. The *tiedTo* primitive uses spreading when there is no MVM on a destination host. Metaglue accomplishes spreading using a *glue spreader*.

The glue spreader is an extremely light-weight daemon process that runs on each host that a Metaglue MVM can spread to. When an MVM needs to spread, a predetermined socket that the glue spreader listens to is contacted on the host, and a message is passed to instruct the glue spreader to start an MVM which in turn, establishes

a Metaglue Manager agent. This Metaglue Manager agent can then be contacted to start and stop agents on the MVM.

3.4 Keeping It All Alive

We have looked at how to build Metaglue agents in this thesis, and we have examined the Metaglue system agents. We have investigated the environment that supports them and how these agent platforms are established. We saw how through the primitives, system agents, and platforms, agents are created, distributed, and connected. Now we turn our attention to the two Metaglue mechanisms that automatically maintain the connections between the agents and keep the agents up and running. These mechanisms are called respectively, reconnecting and swapping.

3.4.1 Agent Reconnection

Agent reconnection is the mechanism by which Metaglue maintains the connections between agents. If an agent *reliesOn* another agent, it has a handle to that agent through which the two agents can communicate. If the connection between the agents is severed, *whether due to network problems or the fact that the host that the other agent is on crashes or whether the agent itself dies or moves*, Metaglue will try to restore the connection. Failing that, it will attempt to restart the agent needed in the system so that the first agent can continue to perform its services.

Reconnection is invisible to the agent programmer although it may be customized.

Metaglue accomplishes reconnection through intelligent handles called wrappers. When the Catalog agent prepares a handle to an agent in a system of agents, it takes a simple, unintelligent connection to the agent and wraps it in an object which will, for each available method, persist in completing its request.

There are two notions of persistence, measured via "frustration", in Metaglue reconnection - temporal frustration and frequency frustration. Temporal frustration denotes how long the wrapper will wait for a response from a call from a method in another agent. Frequency frustration denotes how many times it will make that call before it determines that the call is not going to be successful. These parameters are adjustable per method. More sophisticated programmatic control is being added to Metaglue in another thesis by Nimrod Warshawsky at the MIT AI Lab.

If the wrapper determines that the call is not going to be successful, it will then attempt to re-rely on the agent in an attempt to establish a fresh connection or start a new agent as necessary. In the worst case, the wrapper will generate errors that may be used by monitoring and debugging tools so that human operators can intervene. Wrappers can also be forced to stop in their persistence by intervening human operators.

3.4.2 Agent Swapping

Agent swapping is the ability to shutdown an agent, change the definition of the agent (i.e. modify the agent's code and recompile), and bring instantiations of the agent under the new definition back into the running system. Along with the reconnect mechanism, swapping keeps the system up and running while individual components are

modified.

The specification (i.e. the interface) of an agent cannot be changed but the implementation of the methods can be changed. The swapping mechanism is particularly useful during debugging for fixing bugs or for enhancing an implementation of an agent. The whole system of agents does not have to come down and come back up for the modification of agents in clusters of agents that have little or no reliance on one another.

Metaglue accomplishes swapping using the shutdown primitive, the reconnect mechanism, and a component called the agent loader[4] for loading new agent definitions into the MVMs. The agent loader searches for a particular agent definition in libraries of agent definitions (so called "agentlands"). The loader then loads the code into the platform and converts it into an agent, replacing the old agent definition.

Just the simple mechanism of swapping agents in and out of a live system of agents proves to be helpful in keeping a system of agents up and running during modification (extending the system and debugging) or just for restarting misbehaving agents.

3.5 Debugging a System of Agents

Debugging in a distributed system is a focus of research in its own right. Metaglue provides some simple first-order debugging aids for effectively debugging a live system of agents, some of which have already been described. The Catalog Monitor and Configurator agents are two such tools. Swapping is used to make debugging

[4] The agent loader replaces the Java system class loader; it is not an agent.

bearable and cost-effective by keeping the system of agents up. Swapping also makes the evolution of agents possible.

Another debugging aid is the Agent Tester agent in conjunction with the Agent Definition Monitor agent. Using these two agents, large libraries of agent definitions can be searched and agents can be launched straight from these definitions (see figures 3.3 and 3.4). Then using the Catalog Monitor agent, the launched agents can be directly controlled in the running system of agents.

For example, in figure 3.3 the user can launch the Lamp Manager agent. The Lamp Manager agent launches Lamp agents for all of the lamps in the room. In Hal, there is one by the door and one by the window. In figure 3.4, the user uses the Agent Tester agent to open a window that lists the methods of the lamp by the door so that it can be manually controlled (i.e. turned on, turned off, dimmed, etc…).

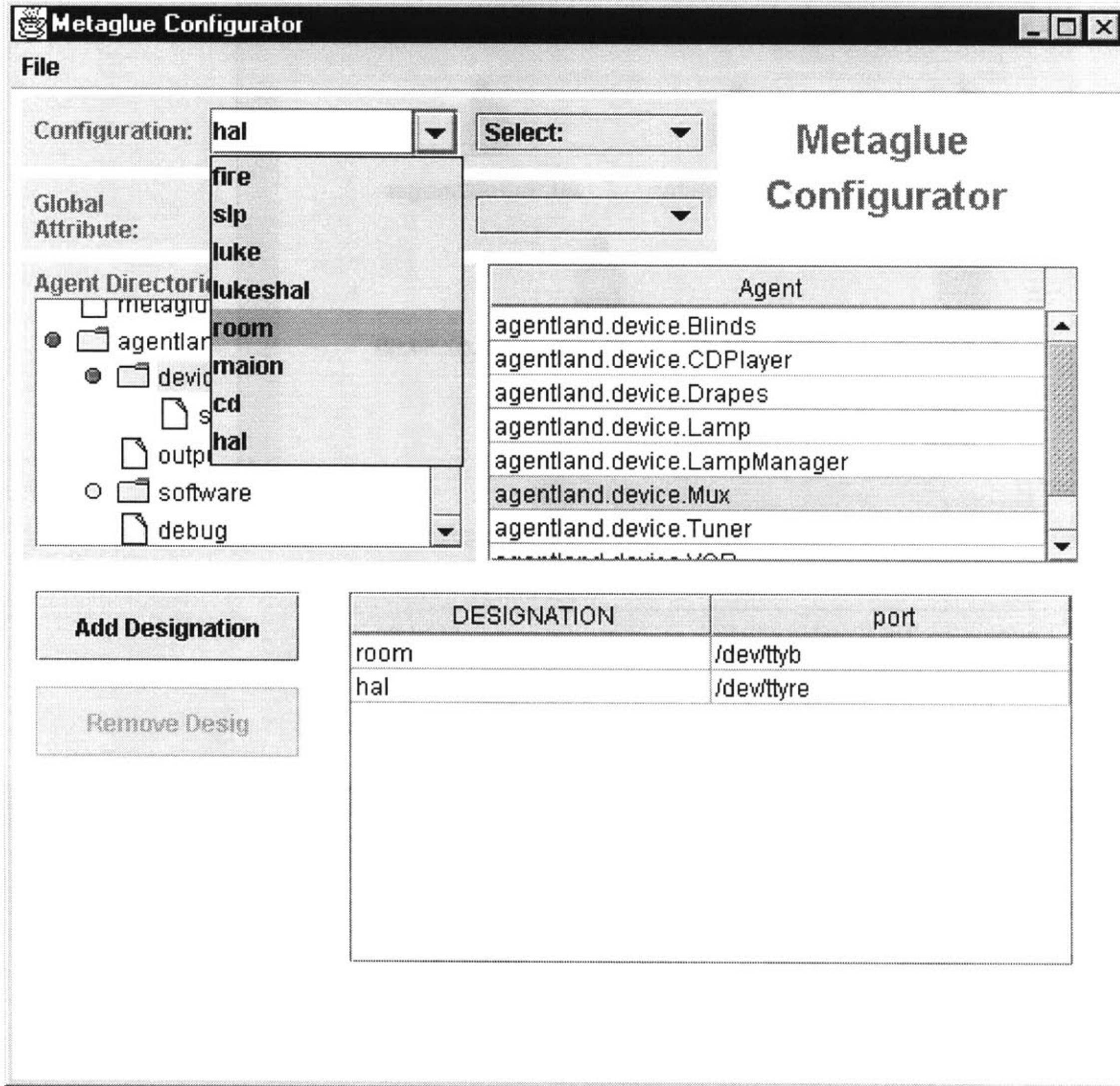


Figure 3.1 – The Configurator agent presents a GUI for browsing the Attribute Database.

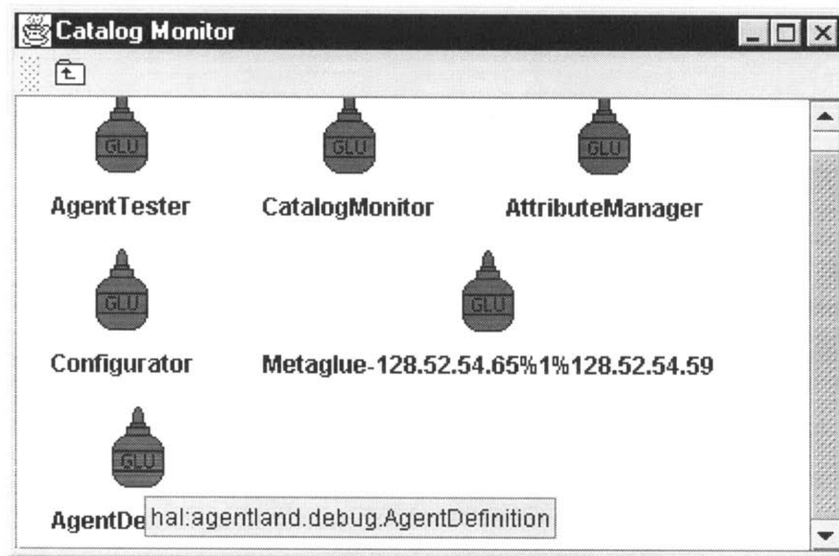


Figure 3.2 – The Catalog Monitor agent presents a graphical front end into the Catalog agent.

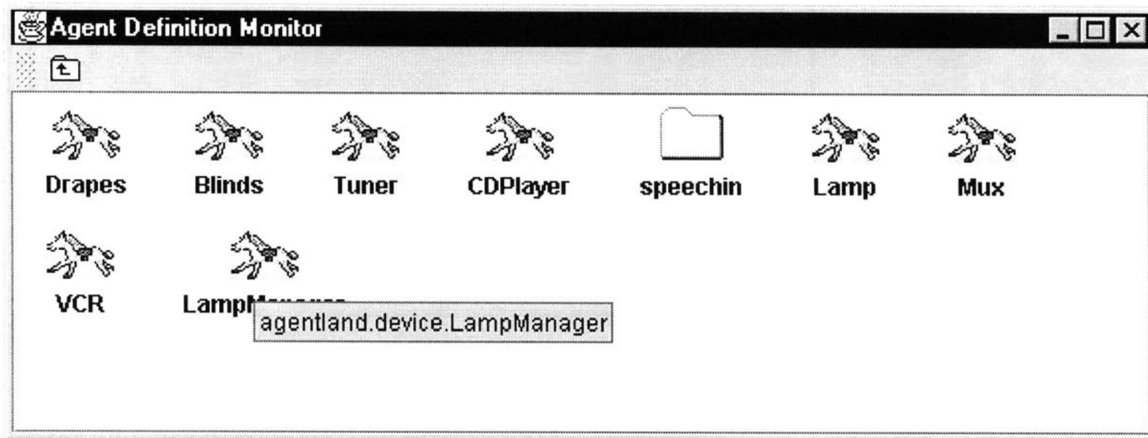


Figure 3.3 – The Agent Definition agent presents a GUI for browsing the available library of compiled Metaglue agents.

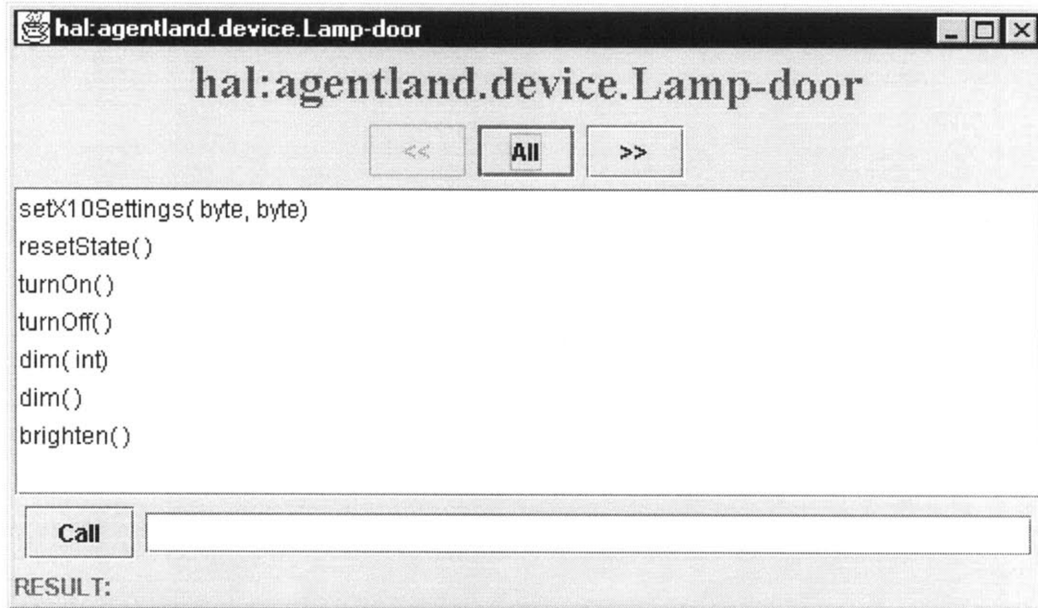


Figure 3.4 – The Agent Tester agent allows for any Metaglue agent to be launched and any method called. Above, a window for controlling the lamp by the door pops up per request.

CHAPTER 4

Conclusions

We conclude this thesis with an assessment of how well Metaglue meets the goals set for it from chapter 1 along with some of its limitations. We will also discuss the design and implementation challenges encountered while building Metaglue. Finally, we will take a peek at some of the future work planned for Metaglue.

4.1 Report Card

Metaglue has been successful at providing the software control system for its target application, Hal. A working Metaglue system allowed us to quickly build a library of agents. Scenarios in Hal appeared shortly after Metaglue became available in the Intelligent Room project. Today, there are multi-modal applications for information retrieval, map exploration, and computer-aided design, and more being developed.

4.1.1 Meeting the Goals

In chapter 1 we established four goals for Metaglue. Now we will analyze Metaglue in light of these goals. The first goal was that Metaglue should establish communication channels between software components that may or may not have been designed to explicitly cooperate. Metaglue's proficiency in connecting non-explicitly

cooperating components is reflected in the variety of agents that can be found in the agent library. There are agents to control consumer electronics, lamps, drapes, web browsers, window managers for a variety of platforms, vision systems, speech recognition systems, and many others.

A particular anecdote pertaining to Metaglue's success in achieving this first goal comes from Boris Katz's START team, which needed to integrate speech recognition, a web browser, and their START system. The interaction between these components in addition to map, wall projection, and laser pointing were accomplished using Metaglue surprisingly quickly. The START team was very pleased with the results, and this subsystem still plays a role in the command post scenario in Hal.

The second goal was that Metaglue should establish and maintain the configuration that each agent specifies in its requirements for operation. Metaglue has been very successful in establishing the system of agents for Hal during start up. The instructions for starting up Hal from a cold-start are under a page long. Under the other systems previously used in Hal, the start-up routine was intricate and usually required a few knowledgeable people working together.

The third goal for Metaglue was that it should permit the introduction and modification of agents without taking the whole system down. In terms of introducing agents, Metaglue does quite well. Starting up a lot of agents all at once proves to be a fast and reliable process. Modifying agents already running in the system is accomplished via the swapping mechanism (section 3.4.2). The agent swapping mechanism in the current implementation of Metaglue is still quite rough (see section 4.1.2 on "limitations") and has not been as widely used as planned. However, the

mechanism has been used to make modifications to agents while keeping the overall system of agents running for a few days.

The fourth goal was that Metaglue should support the debugging of a running system of agents. This was accomplished by designing hooks into the runtime so that the communications between agents could be monitored. Also because all agents ultimately extend from the Agent agent, that primitives are easily added for explicit debugging cooperation, i.e. primitives could be placed in agent definitions to signal check points. Another M. Eng. thesis in the Intelligent Room project focuses on these issues.

4.1.2 Limitations

We will now consider some of the limitations of the implementation of Metaglue. First, Metaglue being written in Java is both a blessing and a curse. Java has permitted Metaglue to be painlessly ported to a variety of hosts. On the other hand, since Java is an interpreted language, it is quite a bit slower than compiled languages such as C and C++. Some of the applications such as the vision processing systems that require intensive, real-time computation cannot be directly integrated into an agent but must be linked in using Java's native method interface. Also, Java causes Metaglue to be single-inheritance.

Most of the limitations of Metaglue relate to the swapping mechanism. Swapping turned out to be very hard to implement using Java because, as of Java 1.2, a class cannot be easily reloaded into a virtual machine. Therefore, a hack was required to accomplish

this[1]. This hack imposes limitations, however. Only agents can be reloaded; any non-agent classes that the agents use (e.g. data structures) cannot be modified and swapped. This hack also creates some minor scoping limitations as well[2].

4.2 Challenges

There were some design and implementation challenges encountered during the creation of Metaglue. The biggest design challenge was deciding what was needed in the system and what was not. It was very tempting to make Metaglue more general purpose or overly flexible. To quell this urge, we developed a library of agents before Metaglue was written. Then we developed a number of simple prototypes of Metaglue, to test the libraries. We made changes to the agents, augmented and modified the prototype Metaglues, and eventually settled on a surprisingly small set of primitives.

Most of the engineering challenges had to do with constraints from Java. One challenge in particular was creating the reconnect mechanism (section 3.4.1). Java provides no hooks for internally trapping exceptions due to severed connections between remote objects. One approach we attempted was to rewrite the underlying Java mechanism for remote communication, but this quickly proved to be unwieldy and forfeited compatibility with future versions of Java. Instead, we used the "wrapper" method described in section 3.4.1.

[1] Essentially, the Java class loader caches the loaded classes by name, and this cache is not exposed to programmers. By extending the Java class loader, the agent loader, and instantiating a new agent loader every time an agent is to be loaded, the cache is always fresh and the system will reload the class.

4.3 The Take Home Lesson

Metaglue's greatest strength as a multi-agent programming language and runtime environment is its simplicity. It is easy to learn, and it is easy to convert existing software components into Metaglue agents. Metaglue's two major primitives, *reliesOn* and *tiedTo*, along with other minor primitives, and the mechanisms of reconnection and agent swapping provide a small but powerful toolkit for establishing and maintaining a system of agents.

One criticism of many agent systems is that they are too feature-packed. For example, many agent systems spend a lot of effort in having agents move with their complete state, including their threads of execution. This feature is not widely used in multi-agent applications, at least in modern applications, and tends to make the system more complicated than it needs to be without adding much value.

Other agent systems try to anticipate all possible needs of agents, and the systems become large and complicated. But the variety of agents and the possible organizations of them are so great that predicting all their needs is impossible. Metaglue provides a small set of usable mechanisms and primitives that we believe are sufficiently expressive and are the most useful for multi-agent systems.

4.4 Future Work

---

[2] Only the "public", "private", and "protected" modifiers are allowed in agents; the Java "package default" is not permitted.

Metaglue currently serves as the software control system for Hal, and other students in the Intelligent Room project are enhancing it. Some enhancements under development include:

- A more powerful debugging system with cooperative debugging primitives
- Remote debugging and monitoring via the web
- Security and privacy issues between agents
- *tiedTo* and *reliesOn* extended to work with more abstract requirements

CHAPTER 5

Bibliography

{Bobick}

Bobick, A., Intille, S., Davis, J., Baird, F., Pinhanez, C., Campbell, L., Ivanov, Y., Schutte, A., Wilson, A. Design Decisions for Interactive Environments: Evaluating the KidsRoom. In *Proceedings of AAAI 1998 Spring Symposium on Intelligent Environments.* AAAI Technical Report SS-98-02.

{Coen99}

Coen, M., Weisman, L., Thomas, K., and Groh, M. A Natural Language Modality for an Embedded Multimodal Environment. Forthcoming, 1999.

{Coen98}

Coen, Michael. Design Principles for Intelligent Environments. In *Proceedings of AAAI 1998 Spring Symposium on Intelligent Environments.* AAAI Technical Report SS-98-02.

{Coen97}

Coen, M. Building Brains for Rooms: Designing Distributed Software Agents. *Proceedings of the Ninth Conference on Innovative Applications of Artificial Intelligence.* Providence, R.I. 1997.

{Coen94}

Coen, Michael. *SodaBot: A Software Agent Environment and Construction System.* MIT AI Lab Technical Report 1493, June, 1994.

{GM}

General Magic. *Odyssey (Beta 2)* Agent System Documentation. http://www.genmagic.com/agents.

{IBM}

IBM. *Aglets* Software Development Kit. http://www.trl.ibm.co.jp/aglets.

{Jennings}

Jennings, N., Sycara, K., Georgeff, M. (eds.) Autonomous Agents and Multi-Agent Systems, Vol. 1, No. 1. Kluwer Academic Publishers. 1998.

{Minsky}

Minsky, Marvin. The Society of Mind. Simon and Schuster, Inc. New York. 1986.

{ObjectSpace}

ObjectSpace, Inc. *ObjectSpace Voyager Core Package Technical Overview (Version 1.0).* December 1997. http://www.objectspace.com/voyager/whitepapers.

{Shafer}

Shafer, S., Krumm, J., Brumitt, B., Meyers, B., Czerwinski, M., Robbins, D. The New EasyLiving Project at Microsoft Research. Microsoft Corporation. 1998.

{Sycara}

Sycara, K., Decker, K., Pannu, A., Williamson, M., and Zeng, D. Distributed Intelligent Agents. *IEEE Expert*, Dec-96.

{Waldo}

Waldo, J. Jini Architecture Overview. Sun Microsystems, Inc. 1998.

{Weiser}

Weiser, Mark. The Computer for the 21st Century. *Scientific American.* 1991.

{White}

White, James E. Mobile Agents. General Magic, Inc. 1995.