Extending the Metaglue Multi-Agent System

by

Nimrod Warshawsky

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Electrical Engineering and Computer Science at the Massachusetts Institute of Technology

May 19, 1999

[June 1999]

Author_____
Department of Electrical Engineering and Computer Science
May 19, 1999

Certified by_____
Barbara Liskov
Thesis Supervisor

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

Extending the Metaglue Multi-Agent System
by
Nimrod Warshawsky

Submitted to the Department of Electrical Engineering and Computer Science

May 19, 1999

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

ABSTRACT

Metaglue is a platform for the development and administration of distributed, interconnected computational elements. It is the development platform for the MIT Artificial Intelligent Laboratory's applications in Hal and the Intelligent Room.

This thesis extends Metaglue to make it more effective and robust. Metaglue now allows computational elements to dynamically reestablish their communications channels. In addition, these elements can now begin asynchronously, and they can depend on each other in a circular fashion.

Thesis Supervisor: Barbara Liskov
Title: NEC Professor of Software Science and Engineering

TABLE OF CONTENTS

LIST OF FIGURES

Number *Page*

ACKNOWLEDGMENTS

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.

Chapter 1

INTRODUCTION

The Metaglue system was assembled as a platform for the development and administration of distributed, interconnected computational elements. It is intended as a way of "managing systems of interactive, distributed computations, *i.e.* those in which different components run asynchronously on a heterogeneous collection of networked computers" [Phillips].

Systems like Metaglue are useful in situations where a large amount of computational power is provided through a collection of networked, distributed hosts. Metaglue is an extension to the Java programming language. It is designed to express the interrelationships of a system of distributed computational elements while minimizing the number of primitives added. By building upon Java, a small number of stable and programmer-friendly primitives was designed and implemented.

Though Metaglue was usable in its initial implementation, it had several limitations. This thesis will discuss how I addressed these limitations by extending the Metaglue system.

My extensions include the ability for computational elements to dynamically reestablish their interconnections and for these elements to initialize themselves asynchronously.

Metaglue development has been heavily influenced by the requirements of the Intelligent Room project in the MIT Artificial Intelligence Laboratory. Most of the testing and research has taken place in the Intelligent Room and its sister project, Hal. Examples will be drawn directly from applications developed in these two environments, and much of the discussion will be laced with Room and Hal references. Therefore, the following section will provide a brief introduction to these projects.

1.1 The Intelligent Room & Hal

The Intelligent Room project in the Artificial Intelligence Laboratory is a research effort that embeds computers in the user's work/living space. The project began as an enhanced office space/laboratory, where a user could interact with computers using non-standard input modalities. The room gets its input from speech recognition and machine vision systems. With an assortment of devices, such as cameras, microphones, and projectors, as well as varied software components, the Room is a distributed headache. The original monolithic C infrastructure proved to be overly burdensome to modify. A subsequently written Perl multi-agent programming language, Sodabot [Coen94], "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" [Phillips].

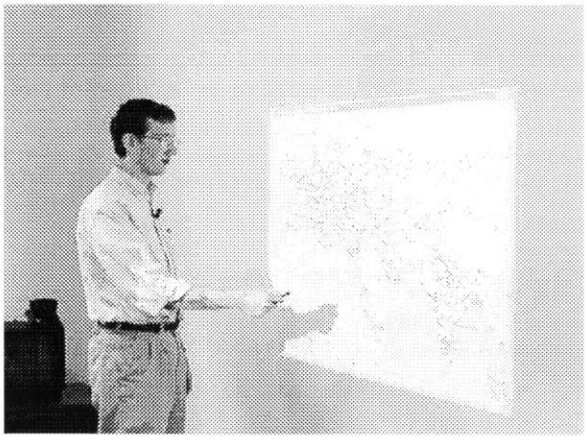
Control of the Intelligent Room's dynamic elements requires vast amounts of processing power. In mid-1997 a new environment was proposed, whose computational requirements dwarfed those of the Intelligent Room. Named Hal, this environment is composed of literally dozes of hardware and software components and simulates a living quarter, where an individual can work, relax, and socialize. Hal has real-time control over its devices, which includes the ability to modify ambient light levels, the state of the blinds/drapes, a stereo system, two projectors, a VCR, and six cameras. A user can interact with Hal using speech, postures (such as lying down, entering/leaving the room), laser pointers, and other input modalities. Hal is connected to software such as Boris Katz's START system, through which a wide assortment of queries can be handled. Administering all of these devices and their interactions requires a sophisticated software and hardware infrastructure.

Figure 1 Hal

A sample application implemented for Hal is called the Command Post of the Future. Hal displays a dynamic map of the world on one projector, and a Netscape browser on another. The map dynamically plots the locations of tropical storms, hurricanes, Mir, and NASA's space shuttle, overlaid on a world map. Using a laser pointer, a user can direct the Command Post's attention to a region and request information or perform an action. For example, the user can use the laser pointer to point at the Balkans region and say aloud, "Computer, zoom in." Hal will zoom in on the region and redraw the map with additional detail (such as cities). The user can ask the Command Post a question, such as "Does Yugoslavia have nuclear missile capabilities?" or, if Yugoslavia has already been selected by the laser pointer, the user can ask, "Does *this* country have nuclear missile capabilities?" The Command Post will redirect the query to START, and Hal will provide the response

using digitized speech. The user can further ask, "What is the weather in Belgrade?" Again, Hal will redirect the query to START and display the result in the Netscape browser.

Figure 2 Command Post of the Future



This application combines the functionality of a dynamic map, a context-sensitive speech recognition system, START, a Netscape browser, a speech synthesis system, two vision systems (to observe the laser pointer on each of the two projection screens), as well as dozens of software components responsible for administering the projectors, the lamps, the VCR, the blinds/drapes, etc... Orchestration of all the various components is clearly an engineering challenge.

The Command Post application demonstrates how Hal can serve as an assistant in a "work" environment. Developers have also written an application that shows how Hal can improve an individual's living environment. To observe the user, multiple cameras have been situ-

ated in the room. Using machine vision, Hal observes when the user lies down on the couch. Under the appropriate conditions, Hal assumes the user is prepared to take a nap, and so it closes the drapes, dims the lights, and plays a user's selection of music (such as Mozart) at low volume. When the user resumes a sitting position on the couch (or rises from the couch completely), the music is stopped, the drapes opened and the light levels are raised. This application can be expanded so that Hal automatically takes phone messages, asks the user for a wake-up time, and records any shows/news the user "usually" watches. Even without the additional functionality, the "resting demo" requires eighty software components running on over a dozen workstations.

The complexity of these scenarios is further heightened by the requirement for real-time responsiveness from the environment. [Coen97] discusses the difficulties in building a monolithic system capable of handling the processing requirements of such systems. Using a distributed multi-agent approach, Metaglue has successfully met the computational and performance demands of systems such as Hal and the Room.

1.2 Multi-Agent Systems

In the field of Computer Science, the term "agent" has come into regular use. Unfortunately, the term has adopted varied meanings. A commonly used definition is a "computational system which is: long lived; has goals, sensors and effectors; decides

autonomously which actions to take in the current situation to maximize progress towards its (time-varying) goals" [Maes]. We have found that agents of this type are overly complex for our needs and poorly express the interdependencies of an environment like Hal.

The term "agent" in the context of multi-agent systems is quite different. Our use of the term originates from [Minsky]. In *Society of Mind* the term is used as a computational element that is conceptually simple and has no inherent "intelligence." By connecting these simple agents, complex and "intelligent" behaviors emerge. In Hal, eighty very simple software agents are connected, resulting in a system with sophisticated behaviors. Minsky argues that the human mind's intelligence can be similarly "constructed" by connecting unintelligent agents.

A specialized, computationally simple element that interacts with similar, independent elements is an excellent way of expressing the relationships among components in Hal. As long as an agent has a means of communicating/interconnecting with other agents, they are physically independent. In Metaglue, barring a physical dependency on a machine, *e.g.* a vision agent must run on a machine equipped with a frame grabber, an agent can run on any host, as long as both hosts can communicate with each other over a network.

A discussion of Hal's "resting demo", where the user lies down on the couch, will show how Minsky's concept of agents has been used in Metaglue.

1.3 RESTING DEMO

The "resting demo" involves the cooperation of most of the computer-controllable devices in Hal. How these devices are controlled will now be discussed.

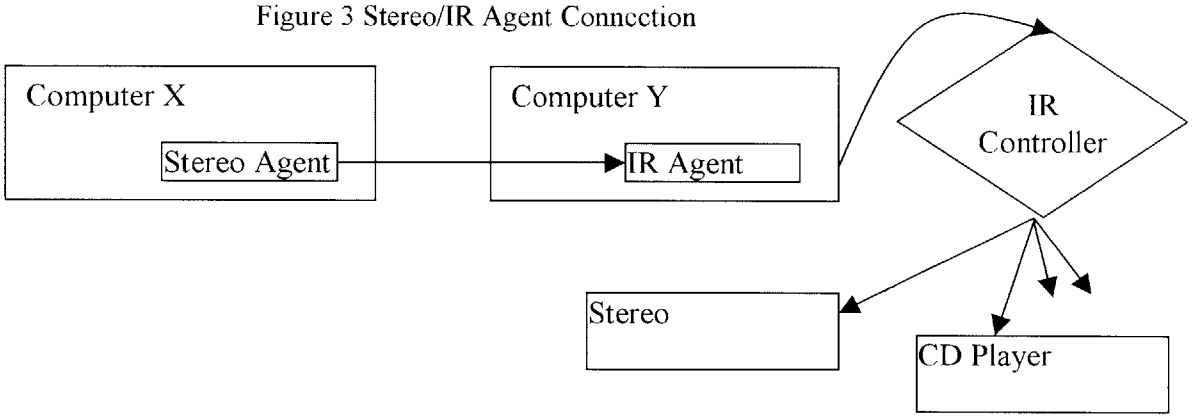
Devices in Hal fall into one of three categories, those that can be controlled using a serial port, those that can be controlled using X-10[1], and those that must be controlled through their IR receiver. Hal has two motorized, steerable cameras that can be controlled by issuing commands from a computer's serial port. The lamps in Hal are controlled using X-10 modules. However, to control devices such as the VCR, HAL must simulate the VCR's remote-control IR signals. This is accomplished by using an IR controller device connected to a computer's serial port. The IR controller works by storing the composition of an IR signal in assigned banks. These signals can then be regenerated through IR emitters that are placed over the appropriate device's IR receiver.

When the user begins the "resting demo", agents responsible for the cameras, the lamps, the stereo, speech recognition, speech synthesis, X-10, and IR are started. Each of these agents will move to the appropriate computer. The camera agent will automatically move to the computer with a frame grabber, the X-10 agent moves to the computer connected to the X-

[1] An X-10 controller is connected to a computer, and transmits signals over a room's electrical wiring. Devices such as lamps are plugged into X-10 modules on the same circuit. The computer can issue instructions through the X-10 controller asking the X-10 module to close the switch (turning the lamp on), to adjust the current flow (dimming the lamp), or to open the switch (turning the lamp off).

10 controller, the IR agent moves to the computer connected to the IR controller, the speech recognition agent moves to the computer running IBM's Via Voice, and the speech synthesis agent moves to the computer running the Laureate speech synthesis system. When the vision agent sees a person lying down on the couch, the lights agents are instructed to dim the lights, the stereo agent begins playing music, speech recognition listens to the user, and output is directed through the speech synthesis agent.

Figure 3 Stereo/IR Agent Connection



Each agent is responsible for a relatively simple task. For example, the lamp agent needs only know how to interact with the X-10 agent to modify the state of the lamp. With minimal additional logic, an application like the "resting demo" can be implemented by connecting these conceptually simple agents.

1.4 Design Goals

In this section I will discuss the specific goals we viewed as important while designing Metaglue. From experience with the two prior multi-agent systems used in the Intelligent Room, certain goals emerged as essential. These are:

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 [Phillips].

A multi-agent system, called Metaglue, was implemented with these design goals in mind. To better understand Metaglue, as well as my extensions, I will now discuss each design goal in turn.

1.4.1 Establishing Communication Channels

> 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 with each other [Phillips].

The fact that Hal is complex and computationally intensive should be clear, though the remaining engineering challenge requires some elucidation. Communication between agents on different hosts, using different operating systems is a challenge in and of itself. To compound the difficulty, Hal requires that certain software components, such as Netscape Navigator on a Solaris based computer, communicate information with a Windows 95 based speech recognition system. These software components were not designed to work with each other. These communication channels are the computational "glue" that ties the agent system together.

Using Java, the original implementation of Metaglue was very successful in realizing this goal. Java's inherent platform independence and Remote Method Invocation (RMI) protocol are a strong foundation upon which to build a multi-agent system.

1.4.2 Establishing and Maintaining Agent Configuration

> 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...the second agent needs to exist and be functional.
>
> ...
>
> Not only should Metaglue establish the agents and their interconnections, but also it should attempt to maintain them [Phillips].

The original Metaglue platform was successful in establishing agent interconnections, however provided no mechanism for the preservation of these connections. If an agent fails, or the programmer stops it and restarts it, Metaglue could not automatically reestablish the connections. In extending Metaglue, I have provided a means for persistent agent connections through dynamic reconnections. Among other extensions, this thesis will discuss how agent connections are preserved when communication failures occur, when agents move from one host to another, or when an agent is stopped and restarted.

1.4.3 Dynamic Agent Loading

> Metaglue is designed to permit the introduction and modification of agents without taking the whole system down.
> ...
> New agents may be introduced to control newly introduced hardware and software [Phillips].

Metaglue has successfully provided the agent programmer with this functionality by building upon Java's Reflection API. Hal can be extended dynamically without having to shut the system down, and new functionality is often introduced in this manner.

1.4.4 Debugging a Running Multi-Agent System

> A running system of Metaglue agents should be debuggable...However, debugging agent systems requires non-traditional debugging strategies [Phillips].

Introducing an effective debugging mechanism to Metaglue is the same task as developing an effective debugger for a (possibly massively) distributed system, an area of continuing research, both in the Intelligent Room project and in the larger multi-agent system research community. At this time, an interactive debugger has not been integrated. Metaglue does contain a program called AgentTester that is used to start agents and observe any exceptions generated. This falls short, however, of a general-purpose debugger.

1.5 METAGLUE EXTENSIONS

This thesis will describe the extensions to Metaglue along three principal axes. The first is modifications to the infrastructure of Metaglue. The second involves giving agents the ability to dynamically reestablish communication channels. The third involves allowing agents to start asynchronously.

The Metaglue infrastructure was simplified conceptually. Components of Metaglue that made sense at design time, but proved unsatisfactory, were modified or removed.

Dynamic agent reconnection is a vital capability of a real-world system. In an environment as complex as Hal, components of the room occasionally fail or are restarted. I have given Metaglue the ability to reestablish the communication channels to these components. By

automatically intercepting an agent's method calls, Metaglue can attempt to resolve any communication problems.

Prior to asynchronous agent startup, circular agent dependencies were not allowed in Metaglue. This was an artifact of a faulty implementation, which I corrected.

The remainder of this thesis will give a thorough introduction to Metaglue, and a discussion of my modifications. Chapter two will discuss Metaglue in detail, while Chapter three will provide the reader with an overview of the above mentioned extensions. Chapter four will describe the implementation details of these extensions, and Chapter five will conclude the thesis, and describe areas for future work.

Chapter 2

METAGLUE

The Metaglue system is an extension of Java 1.2. Only a superficial knowledge of Java will be required to understand this introduction to Metaglue. Later chapters will delve into the implementation details of the Metaglue system and its extensions, and as such, comfort with Java and its Remote Method Invocation (RMI) mechanism will be required.

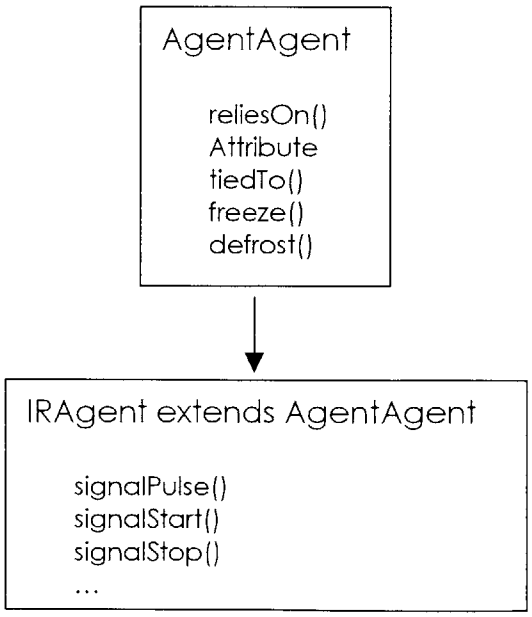
Metaglue has introduced three new primitives (as well as an assortment of supporting functionality), tiedTo(), reliesOn(), and freeze()/defrost(), to the Java API. The behaviors of these primitives serve as the "glue" that ties Metaglue agents together. Using these primitives, and Java's large API and platform independence, Metaglue has successfully provided an infrastructure capable of handling the computational requirements of a system as sophisticated as Hal. More than half a dozen programmers have developed applications for both Hal (the Command Post of the Future, the "resting demo", etc...), as well as the Intelligent Room. Metaglue has proven to be a stable, efficient, easily maintainable, and effective platform for the development of multi-agent systems.

The following section will introduce Metaglue's primitives as presented in [Phillips], along with several subsequent modifications that are not directly part of my thesis.

2.1 PRIMITIVES

A Metaglue agent is simply a Java class that inherits from the AgentAgent super class. This class provides the agent with the primitives required for successful operation in a Metaglue system. A Metaglue agent is identified by an AgentID that is composed of a society, an occupation and a designation. The occupation of an agent is what task the agent is responsible for, *e.g.* VCR, IR, X-10, AgentTester, Lamp. The AgentID will be further discussed in the section on Supporting Infrastructure.

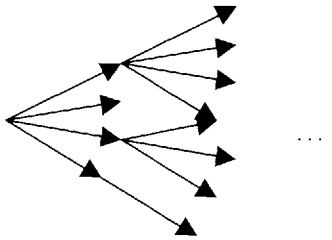
Figure 4 IRAgent inherits from AgentAgent



2.1.1 RELIESON()

The reliesOn() primitive is used to establish dependencies, as well as to provide a communication channel between agents. This method effectively serves to define a directed graph of agent dependencies. In the original implementation of Metaglue, cyclic agent-dependencies were functionally disallowed. This was an implementation deficiency that was corrected, and will be discussed in the section on asynchronous reliesOn(). Reliance can only be established between Metaglue agents, *i.e.* an agent can not rely on a computer, a device, an object, or a value. A reliesOn() invocation can generate a flurry of activity as the dependence graph expands.

Figure 5 Directed Graph of Agent Dependencies



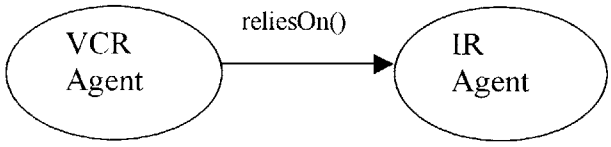
The simplified syntax of a Metaglue reliesOn() is as follows:

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

An expanded version of the above statement is used in Hal to establish a connection/dependence between the calling agent (in this case VCR) and the IR agent. If the IR agent is not already running, reliesOn() will start the agent. The object returned from

reliesOn() (ir in the example above) can be used as a "handle" to the target agent, and method invocations can be issued using standard Java notation: ir.signalStart(...).

Figure 6 VCR/IR Reliance



In the Phillips implementation, the handle returned by reliesOn() had no error-handling/reconnect capabilities. If an agent was shutdown and then restarted (called agent swapping), any objects that had a reference to the agent would fail on the next method invocation. I will discuss how I improved the system by introducing dynamic agent reconnection later in this thesis.

reliesOn() is a method that belongs to the AgentAgent[2] class. Every agent inherits this functionality. Calls to reliesOn() are "local" versus "remote", they are executed using local method invocations, rather than Java's remote method invocations (RMI). All inter-agent communication uses RMI, since every agent is defined to be its own remotely callable object. This fact is crucial for agent reconnection, and we will return to it in the section on dynamic agent reconnection.

---

[2] Every Metaglue agent is comprised of an interface definition file (a Java interface), as well as the actual agent code. The interface definition file is formed by assigning it the occupation of the agent, *e.g.* VCR.java. The actual agent code file name (and therefore the class name as well) is formed by taking the occupation and appending the word Agent, *e.g.* VCRAgent.java. This convention was maintained throughout the system, and AgentAgent is the implementation of the Agent interface.

A semantic rule of the Metaglue system requires that agent dependencies be established using the reliesOn() primitive, *i.e.* agent X may not pass its reference of agent Y to agent Z. If communication of this sort is necessary it should be conducted using an agent's AgentID, *i.e.* agent X should pass agent Z's ID to agent Y. As an example agent X can pass the requisite information to agent Y using:

```
agentY.useThisAgent(agentZ.getAgentID());
```

Whereas the following is disallowed: `agentY.useThisAgent(agentZ);`

2.1.2 ATTRIBUTE

An attribute is a way for an agent programmer to conveniently store dynamic characteristics of an agent. The attribute is stored in a persistent SQL database, and its value can be changed without modification to an agent's source code. Attributes are meant to describe an agent, not its state. For example, if an agent opens a socket connection, the socket number can be stored as an attribute. If this value changes in the future, the programmer need only change the value in the SQL database, without modifying and recompiling the agent source code.

An attribute is described by the ID of the agent that owns it, as well as a string, such as "socket" or "port". Therefore, the same string can be used in different agents without conflict.

Here is a way to retrieve the port value from the Attribute database into a variable:

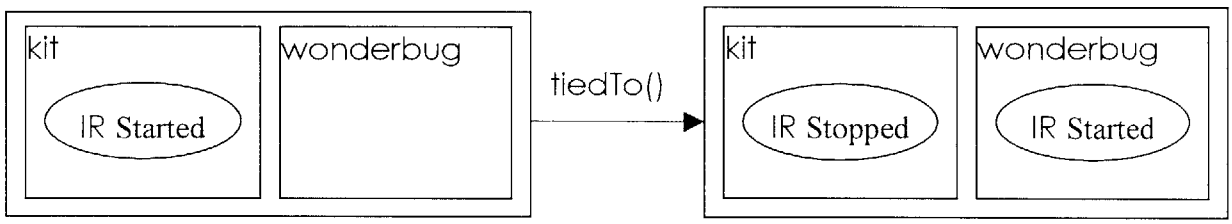
```
Attribute port=new Attribute("port");
int portID=port.getValue();
```

2.1.3 TIEDTO()

The tiedTo() primitive is used to notify the system of an agent's physical reliance on a specific host. For example, the IR controller device is connected, through a serial cable, to a workstation in the lab whose hostname is wonderbug. Therefore, the IR agent can only execute on wonderbug. If the IR agent is started on another machine in Hal, such as kit (another computer used for Hal), it should be automatically shutdown and restarted on wonderbug. The tiedTo() primitive allows the agent programmer to handle environmental/physical constraints such as this. If an agent is not explicitly tied to a specific computer, then it is possible for that agent to run on any one of the hosts that have been registered with the Metaglue system. Furthermore, if an agent that has been explicitly tied to a specific machine is started on a different machine, the Metaglue system will shut the agent down on the local machine and will dynamically restart it on the correct machine.

Though dynamic load balancing can be performed using this infrastructure, this capability is presently lacking.

Figure 7 IR Agent startup with tiedTo()



Like reliesOn(), tiedTo() is a method inherited from the AgentAgent super class, and so every agent has this functionality. An invocation of tiedTo() should be performed in the agent's constructor, prior to any actions that may cause side effects. This restriction arises because Metaglue may (if the agent is running on the wrong host) shut the agent down and then begin a *new* instance on the appropriate host. If the IR agent were started on host kit, then it would be shut down and a new instance started on host wonderbug. It should be noted that if the current host has the same name as the hostname passed as an argument to tiedTo(), the method will return normally and commands following the tiedTo() will be executed sequentially. If the hostnames differ, these commands will never be reached in *this* instance.

The fact that an agent may be shut down and restarted means that some work may be performed twice. For example, if variables are given default values, the evaluations will be

performed twice. Similarly, if commands are executed prior to the tiedTo(), they will be executed twice if the agent needs to be restarted.

The IR agent constructor has been slightly simplified, but effectively appears as follows:

```
public IRAgent() {
  Attribute host=new Attribute("host");
  tiedTo(host.getValue());
  ...
```

The process of moving to a different host is called spreading, and requires that a registered MetaglueAgent exist on the target machine, *i.e.* in the IR example, Metaglue must be running on wonderbug at the moment of the tiedTo() invocation. If this is not the case, tiedTo() will block, waiting for the user to start Metaglue on the appropriate machine.

2.1.4 FREEZE()/DEFROST()

freeze() and defrost() are new primitives added to the Metaglue system that allow for persistent storage of volatile information such as field values, state, etc... Using a SQL database, agents now have the ability to store any serializable objects for future retrieval. This functionality is useful in preserving the state of a device. An agent responsible for the light levels of the lamps in Hal can save this information and retrieve it the next time the agent is instantiated.

In pseudo-code, here is a skeletal implementation of the VCR agent in Hal

(Note: this code will not compile)

Figure 8 VCRAgent.java

```
public class VCRAgent {

  private IR ir;
  public boolean powerStatus;
  public boolean muteStatus;

public VCRAgent() {
    ir = (IR) reliesOn("IR"));                          //reliesON
    powerStatus = defrostBoolean( "power" );            //defrost
    muteStatus = defrostBoolean( "mute" );
  }

public void setPowerStatus(boolean on) {
    powerStatus = on;
    freeze( "power", powerStatus );                     //freeze
  }

private void press(String b) {
    send(b);
  }

 private void send(String b) {
   ...
     ir.signalPulse(signalOf(b));       //inter-agent communication
     if(b.equals("POWER"))
         setPowerStatus(!powerStatus);
  }

public void togglePower() {
    press("POWER");
  }

public boolean getMuteStatus() {
    return muteStatus;
  }

  public boolean getPowerStatus() {
    return powerStatus;
  }


  public void turnOn() {
    if (!powerStatus) press("POWER");
  }

  public void turnOff() {
    if (powerStatus) press("POWER");
  }

  public void rewind() {
    if (!powerStatus) press("POWER");
    press("REWIND");
  }

  public void fastForward() {
    if (!powerStatus) press("POWER");
    press("FAST FORWARD");
  }

  public void play() {
    if (!powerStatus) press("POWER");
    press("PLAY");
  }

...
}
```

2.2 SUPPORTING INFRASTRUCTURE

Besides the described primitives, Metaglue also provides supporting infrastructure that identifies agents, allows agents to find each other, and identify computers as Metaglue capable. Each of these capabilities will now be described.

2.2.1 AGENTID

The namespace of Metaglue agents is broken into three groups: society, occupation, and designation. An agent's society indicates what "universe" the agent belongs to, *e.g.* our project has a hal society for Hal, and a lab society for the Intelligent Room. An agent's occupation is self explanatory, examples being IR, VCR, Lamp, etc... Finally, an agent's designation differentiates between agents that belong to the same society, and have the same occupation, but refer to different instances. Inside the Hal environment, there are two computer-controlled lamps. The first is adjacent to the window, and the second is situated next to the door. Light levels can be controlled independently for the two lamps, however from a programmer's perspective they behave identically. Therefore, the agents that control these lamps are identical *except* for their designations, and corresponding X-10 module.

The AgentID is expressed in human-readable form as:

society:occupation-designation

and the two lamp agents have the following names:

hal:Lamp-door
hal:Lamp-window

2.2.2 *METAGLUE CATALOG*

The Metaglue system requires a centralized repository cataloging which agents are running. This is accomplished through a catalog agent that builds directly upon Java's RMI registry capabilities. A Metaglue catalog can handle three basic requests, add(), lookup(), and remove().

When a programmer issues a reliesOn(), Metaglue first checks if the agent exists in the catalog. If it does not, Metaglue will start the agent and add it to the catalog. If the agent already exists in the catalog, Metaglue simply returns a reference to the agent.

A single catalog is meant to be a resource shared across societies, *i.e.* multiple societies can share a single catalog, and there can be only one catalog per host. In the original implementation of Metaglue, a catalog could share a JVM with other Metaglue agents. For reasons that will be discussed in the following chapter, this has been disallowed.

2.2.3 METAGLUEAGENT

The MetaglueAgent is responsible for administering dynamic aspects of Metaglue. For a Java Virtual Machine (JVM) to have the capability of running agents, a MetaglueAgent must be running inside the JVM. The combination of a MetaglueAgent running in a JVM is called a Metaglue Virtual Machine (MVM).

A MetaglueAgent has three primary roles:

1. Starting the Metaglue system
2. Starting agents
3. Identifying hosts as having Metaglue capabilities

MetaglueAgent has a main method that accepts arguments consisting of the appropriate society, the hostname of the catalog host, and an agent to run. The following command is how one would start Metaglue and run the VCR agent in society hal, with a catalog on wonderbug.

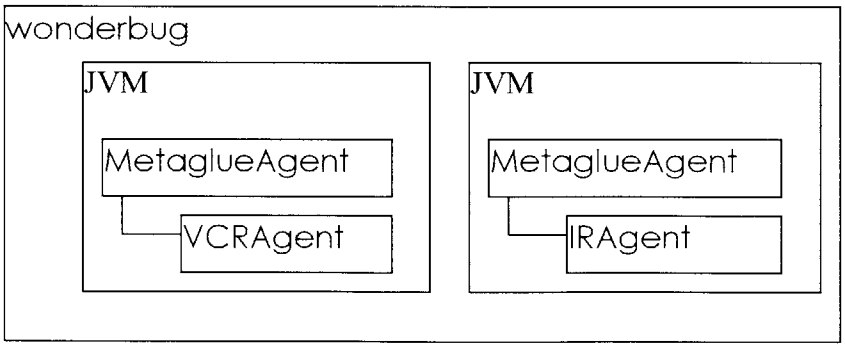
java metaglue.MetaglueAgent hal wonderbug agentland.device.VCR[3]

A computer is identified as having Metaglue capabilities when there is a MetaglueAgent running on the host. Multiple instances of the MetaglueAgent can

[3] The metaglue and agentland.device are Java package names required for identifying components of Metaglue and Hal, but will not be explored further in this thesis.

run simultaneously on a single computer in different JVMs, but there can be only one MetaglueAgent per JVM. Multiple MVMs on a host is useful if multiple programmers are running Metaglue agents on the same computer. In the Phillips implementation of Metaglue, if multiple MetaglueAgent instances were running on the same host, only one of the instances would be registered with the Metaglue catalog. The registration of the MetaglueAgent with the catalog is important because agents can only get a handle to registered instances. Those agents running in unregistered MVMs are still accessible to other agents, but new agents can not be started on these MVMs. This behavior has been modified so that all MetaglueAgent instances are registered with the Metaglue catalog. This is a relatively minor point, and one need only remember that there is a one-to-one mapping between MetaglueAgents and JVMs, while there is a many-to-one mapping between MVMs and computers.

Figure 9 Multiple MVMs on wonderbug

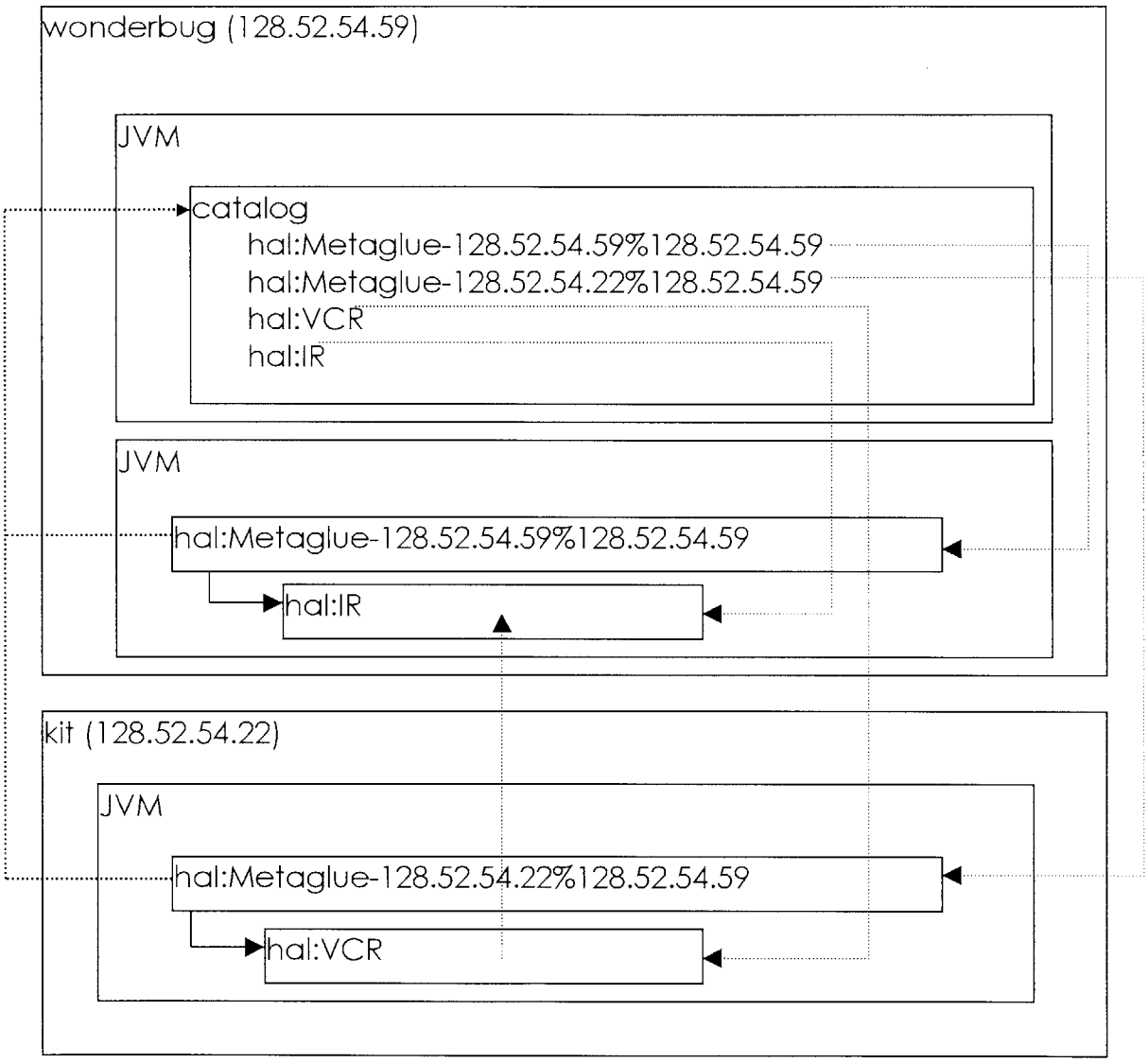


MetaglueAgent has its own naming representation different from normal AgentIDs. A MetaglueAgent is identified by its society, the IP address of its host, and the IP address of its catalog. For a MetaglueAgent running on kit, and using the catalog on wonderbug the ID is:

hal:MetaglueAgent-128.52.54.22%128.52.54.59{d782a53e77}

where 128.52.54.22 is the IP address of kit, while 128.52.54.59 is the IP address of wonderbug. The remainder of the ID ({d782a53e77}) is a unique identifier that differentiates between MVMs running on the same host that belong to the same society, and use the same catalog.

Figure 10 A Running Metaglue System.



Chapter 3

EXTENSIONS TO METAGLUE

Most of the extensions to the Metaglue system have been "under-the-hood", and so a detailed discussion of the implementation specifics will be undertaken. Unlike the previous chapter, a strong grasp of Java will be required to fully understand this discussion.

My extensions to the Metaglue system lie in three domains:

1. MVM modifications
2. Dynamic agent reconnection
3. Allowing circular agent dependencies

3.1 MVM MODIFICATIONS

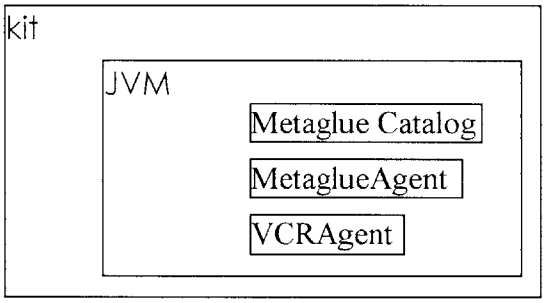
Three substantial modifications were performed to the Metaglue infrastructure:

1. Providing the Metaglue catalog with an independent JVM
2. Removing Glue Spread
3. Allowing the catalog registration of every MetaglueAgent

3.1.1 Catalog Separation

In the original implementation of Metaglue, the catalog was not an independent component of the MVM, *i.e.* it was not a distinct entity. This allowed for situations where the JVM on a host machine would include a Metaglue agent, a user's agents, as well as a Metaglue catalog.

Figure 11 JVM with a catalog and agents



Experience has shown this to be a poor design choice. Because the Metaglue catalog is a resource shared between users and societies, it is undesirable for a user's programs to be running on the same JVM. If a user decides to shut down the JVM that is serving as his MVM as well as his catalog, both will be lost. Any other user that was using this catalog would eventually come across irrecoverable errors. In an environment where debugging of agents is a continuous activity, and over six agent programmers are working at once, programmers are continuously starting and stopping entire MVMs. The destruction of the shared catalog was not a rare occurrence.

It became apparent that a better implementation would give the catalog its own JVM, without a registered MetaglueAgent. Lack of a registered MetaglueAgent ensures that the catalog's JVM remains "pure", without user's agents or programs. Termination of the catalog's JVM would have to be deliberate and the reasons clearly explained to the other programmers. Though this change was relatively simple to implement, it was conceptually and practically important. This modification meant that a catalog would have to be explicitly started and could not be dynamically initiated if Metaglue detected that one was not running. This was accomplished through a command line argument.

java metaglue.MetaglueAgent –catalog starts a catalog on the current host. Note that though a MetaglueAgent is started, it's only purpose is to start the catalog, and does not register itself.

3.1.2 Removing Glue Spread

Coupled with the decision to separate the MVM from the Metaglue catalog was the decision to remove the glue spreader. Phillips describes the Metaglue glue spreader as follows:

> 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 lightweight 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 [Phillips].

The glue spreader daemon process proved to be an unnecessary aspect of the Metaglue system. In an effort to achieve conceptual and implementation simplicity, the distinction between a MVM and a glue spreader was removed. For an agent to run on, or spread to a host, it is required that a MVM be running and registered with the catalog. From the user's perspective, the distinction is invisible; the user types `mg` instead of `glue` at the command prompt on the desired host.

In practice, it was found that an unused MVM is not an excessive burden on the machine. Therefore, the "light-weight" reasoning behind the glue spreader proved to be overly cautious. The conceptual simplicity gained by removing the glue spreader far outweighs the slight performance degradation introduced.

3.1.3 MetaglueAgent Registration

As described earlier, if multiple `MetaglueAgents` were running concurrently under the same society, on the same host, only the first instance created would register itself with the catalog. The reason for this was that agent programmers should not have to be cognizant of

the fact that multiple MVMs exist on the machine. Therefore, when asking for a handle to a MetaglueAgent, simply describing the agent's society, host and catalog should be sufficient.

How then should Metaglue differentiate between the different MVMs? I added a unique identifier to each MetaglueAgent. When the agent registers itself in the catalog it stores this unique ID. However, when the programmer asks for a MetaglueAgent from the catalog, the catalog will return the first functioning agent to have registered itself, irrespective of the unique ID. Therefore, the programmer need not concern himself with knowledge of how many MVMs exist on a host, or what their unique IDs are.

3.2 Dynamic Agent Reconnection

As discussed in the first chapter, the original version of Metaglue was unable to persistently "maintain the configuration that each agent specifies in its requirements for operation" [Phillips]. If the communication channel between two agents was severed, Metaglue had no automated away to restore the connection.

3.2.1 AGENT-METHOD INVOCATION

The manner in which methods exposed by Metaglue agents are invoked is a compromise between readability and run-time control. Three possible approaches to agent method invocation were considered:

1. agentInvoke(myAgent.method, parameters);
2. myAgent.agentInvoke(method,parameters);
3. myAgent.method(parameters);

The first and second forms introduce agentInvoke as a new primitive added to the Metaglue system. The third possibility is the more "natural" form with which Java programmers are already familiar, and was the one chosen.

The first two approaches to agent method invocation make a clear distinction between method calls targeted at agents and those for non-agent objects. This is a somewhat artificial and undesirable arrangement. Nevertheless, either of the first two approaches would allow for run-time control of agent method invocations, *e.g.* agent reconnection, traffic flow control, etc....

At design time, it was decided that agent calls and normal method calls should appear to be identical to the agent programmer. Metaglue allows the programmer to connect agents without knowing where they are running, or if they are remote or local. Therefore, "network" calls should be no different than local calls. Aesthetically, this approach makes

agent code appear clean and elegant, agents are treated as regular Java objects. However, for any run-time control to be introduced, we had to find a way of intercepting method calls targeted at agents.

The most pressing need for dynamic agent reconnection was introduced with the addition of agent swapping. Agent swapping is the ability to shut an agent down, and replace it dynamically with a new instance. This is useful when debugging an agent and the programmer does not wish to shut down all other agents, simply to introduce a new version of an already running agent. Clearly, any agent that has a reference to the original instance of the agent will fail when trying to its methods. Agent swapping does not make sense unless the communication channel between the two agents could be dynamically restored.

Dynamic agent reconnection is vital to any real-world agent system. Even when the programmer does not intentionally terminate an agent, the agent may crash, or network traffic may be such that communication may be suspended for a period of time. For the Metaglue system to be robust, as well as to provide greater flexibility, agent connections must be allowed to reestablish themselves dynamically. It would be ideal for the reconnection to be transparent to the agent programmer, while allowing him to maintain some control over the reconnection system.

Because agent communication is built upon Java's RMI capabilities, we can introduce our error handling logic into the RMI protocol. An original attempt at introducing reconnect logic involved modifying Java's RMI code and incorporating the logic into the generated stub files. This is a poor choice because Java's RMI compiler can change the format of its output in future versions. This solution is also exceedingly dependent on the specifics of Java's volatile RMI specifications. The solution we chose was to generate a new, intermediary class, that would intercept method calls, and pass them on to the stub, while performing all the necessary error catching/correction logic for the programmer.

3.3 Circular Agent Dependencies

In the Phillips implementation of Metaglue, reliesOn() was a synchronous (blocking) operation, *i.e.* a call to reliesOn() does not return until the constructor of the agent being relied upon completes. Most agent dependencies are established in an agent's constructor, because this is where almost all setup/initialization is performed.

What happens if a cyclic dependency of agents is declared, *e.g.* agent X depends on agent Y which itself depends on agent X? When agent X invokes reliesOn() with agent Y's ID as an argument, reliesOn() performs a catalog lookup() to see if the agent already exists; let's assume agent Y has not been started by another agent or user. The catalog lookup() fails, and reliesOn() needs to start the agent. To start an agent, the MetaglueAgent is

notified, and it creates a new instance of the class corresponding to this agent. Only when the class has been instantiated, *i.e.* the agent constructor has completed, does the MetaglueAgent register the started agent with the catalog.

Now we return to our original question. What will happen if in agent Y's constructor it invokes reliesOn() with agent X's ID? Let's assume that both invocations of reliesOn() take place during the constructor of the corresponding agent. When agent Y invokes reliesOn(), the MetaglueAgent has not registered agent X with the catalog (agent X is still in its constructor waiting for the reliesOn() on agent Y to terminate). So when agent Y invokes catalog lookup for agent X, the call will fail. reliesOn() will now start a new instance of agent X. Clearly, this process continues *ad infinitum*.

What if reliesOn() stores a marker in the catalog notifying other agents that in fact the agent is in the process of being instantiated? If reliesOn() was simply a means of expressing agent dependencies, then this proposal would solve the problem. However, reliesOn() both declares dependence, and opens a communication channel. Therefore, reliesOn() must return a handle to the target agent. What should reliesOn() return if the agent is still being initialized? Metaglue had nothing to return, and so could not allow this situation to occur. The solution to this problem in the initial version of Metaglue was to state that any dependencies that *might* result in a circular dependence could only be de-

clared in the body of the agent, after it has been registered with the catalog. Clearly this was a deficiency in the implementation of the system.

A solution to this problem introduced itself once the intermediary classes from dynamic agent reconnection had been added to the system. By making the intermediary classes "intelligent", they can be returned from a reliesOn() invocation before the agent instantiation has completed. These intelligent wrappers will only block when the agent's methods are invoked. Therefore agent startup becomes an asynchronous process, and circular agent dependencies can be allowed, fixing the implementation flaw. The programmer still needs to beware of a deadlock situation in which both agents rely on each other and also make method calls to each other, in their constructors. The method invocations will block, and neither agent will ever complete its constructor.

Chapter 4

EXTENSION SPECIFICS

I will provide a brief introduction to Java's RMI before discussing the specifics of the extensions I made to Metaglue. This will serve as a refresher for people who already have programming experience with the API. For others, a reading of [Horstman] or the Java Remote Method Invocation Specification is highly recommended.

4.1 JAVA RMI

Motivation for this section stems from the fact that all inter-agent communication uses Java RMI. For a Java class to be available for remote method invocation, the programmer must process the class appropriately. First, the target class must extend the Java UnicastRemoteObject class. Second, the target class must implement an interface that itself extends the Remote interface. Third, the class must be compiled using the normal Java compiler as well as the RMI compiler. Only those public methods that are declared in the above mentioned interface will be available as remote methods. An example should help clarify this:

Figure 12 Java RMI Example

RSample.java

```
public interface RSample extends Remote {
  public void firstMethod() throws RemoteException;
  public void secondMethod() throws RemoteException;
}
```

RSampleImpl.java

```
public class RSampleImpl extends UnicastRemoteObject implements RSample {
  public RSampleImpl() throws RemoteException {
  }

  public void firstMethod() throws RemoteException {
    System.out.println("Inside firstMethod");
  }

  public void secondMethod() throws RemoteException {
    System.out.println("Inside secondMethod");
  }
}
```

The first Java file is an interface definition by the name of RSample, while the second is a class definition by the name of RSampleImpl[4]. Together these files define an object that can handle remote method invocations. The methods that can be invoked remotely are firstMethod and secondMethod. Every method available for remote invocation *must*, by Java semantics, be declared to throw a RemoteException. This is because method

[4] Metaglue uses the RSampleAgent convention instead of RsampleImpl because it is more descriptive than the vanilla RMI convention.

invocations on a remote object can fail for a variety of reasons, and the RemoteException is the JVM's means for notifying the caller of an invocation failure. Compilation of the file RSampleImpl.java will generate the normal RSampleImpl.class. The programmer must then pass the RSampleImpl.class through the rmic compiler. This compiler will generate a RSampleImpl_Stub.class and RSampleImpl_Skel.class. The stub file acts as a server that starts a new thread with every incoming remote method invocation. Therefore, a single remote object can handle multiple RMI invocations concurrently.

To use RSample from a remote machine, RSample must be registered with an RMI registry. This registry is simply a centralized repository of classes available for remote invocation. To use a remote object, the programmer can perform a lookup() in the registry, which will return a stub. The programmer then uses this returned object as he would any other class reference and invoke methods the same way he would a normal method.

In the event of a communication failure, a RemoteException will be thrown. A programmer must either wrap the remote method invocation in a try/catch block or declare that the calling method throws a RemoteException.

4.1.1 Error Handling

As described earlier, every remote method invocation has the possibility of generating a RemoteException. Since every agent method invocation is in fact a remote method invocation, every agent call can generate this error. For agent programming to be robust, the programmer must wrap every remote invocation with a try/catch block (or the less discriminating approach of placing a single block that catches all remote exceptions). This is a tremendously tedious task, since much of the exception handling logic would be similar, but not necessarily identical across method invocations. In the Phillips implementation of Metaglue, the remote exception was simply ignored, and each method that invoked agent methods was also declared to throw remote exceptions. Phillips understood this to be a temporary solution that required modification.

A remote exception can be thrown for reasons other than network degradation and an agent crash. When agent swapping takes place, the reference to the agent is no longer valid, and method invocations will generate this exception.

Using Java's Reflection API, I have written a system that provides dynamic agent reconnection in a way that is transparent to the agent programmer, while still providing him with control over the reconnection system's behavior. This system uses an intermediary class that is built using the AgentPrimer program.

4.1.2 AGENTPRIMER

AgentPrimer works by building an intermediary class between the calling agent and the called agent's rmic generated stub. For example the VCR agent is composed of the VCR.java interface file as well as the VCRAgent.java file. The only methods that need to have error handling/dynamic reconnection capabilities are those in the interface file, since those are the only methods that can be invoked remotely. Using Java Reflection on VCR.class (the compiled version of VCR.java) the AgentPrimer generates a new class (original interface name appended with the string "EHA" for error-handled agent), *i.e.* VCREHA.java.

The AgentPrimer works by iterating through each of the interface methods, and generating wrappers. These wrappers catch the RemoteException and pass it to an error handler. Below is part of the generated EHA file for Hal's computer-controlled lamps.

Figure 13 LampEHA.java

```
...
public class LampEHA implements  ExceptionHandlerInterface,
agentland.device.Lamp, metaglue.Agent
{
private AgentExceptionHandler aeh;
private Lamp agent;                    //reference to stub
private AgentID agentID;
private MetagluePrimitives mp;
...
public void replaceExceptionHandler(AgentExceptionHandler r)
{
  aeh=r;
}
...

public void brighten() throws java.rmi.RemoteException
{
  boolean repeat;
  if(agent==null)                  //used for asynchronous reliesOn()
  {
    instantiate(mp.reliesOnSynch(agentID,false));
  }
  do
  {
    try
    {
      repeat=false;                //allow for reinvocation
      agent.brighten();            //actual method invocation
    }
    catch(RemoteException e) {
      repeat=aeh.handleRemoteException(e,this,"brighten");
    }
  }
  while(repeat==true);
}
```

```
public void dim() throws
java.rmi.RemoteException
  {
  boolean repeat;
  if(agent==null)
  {
    instantiate(mp.reliesOnSynch(agentID,false));
  }
  do
  {
    try
    {
      repeat=false;
      agent.dim();
    }
    catch(RemoteException e) {
      repeat=
        aeh.handleRemoteException(e, this,"dim");
    }
  }
 while(repeat==true);
}
```

Every EHA class has fields for an AgentExceptionHandler, AgentID, MetagluePrimitives, and the corresponding agent occupation (in this case Lamp) interface. The AgentExceptionHandler encompasses almost all of the error handling logic. As can be seen from the replaceExceptionHandler method (added to every EHA by the primer), the handler can be dynamically swapped for a user-defined implementation, thereby providing programmer control. The agent occupation interface is the EHA's handle to the actual stub, and the method invocation is performed through this object. The remaining two objects are required to perform asynchronous reliesOn(), which will be discussed later in the thesis.

The wrappers simply intercept the agent method invocations and pass a RemoteException to the error handler, described in the following section. The repeat field allows the exception handler to re-invoke the problematic method. Though run-time debugging abilities have not been introduced, it would be a simple matter to modify the wrappers so that they store information regarding method invocations to disk, or notify a central agent whose purpose is to log such information.

4.1.3 AGENTEXCEPTIONHANDLER

The AgentExceptionHandler encompasses the actual error handling logic. It will be easiest to discuss this object after reviewing the source code.

Figure 14 AgentExceptionHandler.java

```
...
public class AgentExceptionHandler
{
 private Catalog catalog;
 private AgentID id;
 private MetagluePrimitives callerAgent;

 public AgentExceptionHandler(MetagluePrimitives ap,AgentID aid,Catalog c) {
  id=aid;
  catalog=c;
  callerAgent=ap;
 }

 public boolean handleRemoteException(RemoteException re,ExceptionHandlerInterface caller,
                                      String method_name) {
  System.out.println(method_name+" Threw Exception: "+re);
  try  {
   catalog.alive();   // Test that the catalog is alive ------------> 1
  }
  catch(Exception e) {
   System.out.println("METAGLUE: Exception handler found the catalog to be dead");
   throw new CatalogAccessError();
  }
  try {
   System.out.println("METAGLUE: Exception handler looking for a working stub in the catalog");
   caller.instantiate(catalog.lookup(id)); ------------> 2,3
  }
  catch(Exception e) {
   System.out.println("METAGLUE: Exception handler attempting a new reliance "+id);
   caller.instantiate(callerAgent.reliesOnSynch(id,false)); ------------> 4
  }
  return(true); ------------> 5
 }
}
```

The code performs the following actions:

1. Test if the Metaglue catalog is alive
2. Perform a Catalog lookup() to check for a working stub[5]
3. If a working stub exists, replace the wrapper's current stub with the working stub from the catalog
4. If a working stub does not exist, perform a reliesOn() (the appended synch refers to a request for a synchronous agent startup)
5. Repeat the method invocation

A lost catalog is considered an irrecoverable error, because all bindings have been lost and reestablishment is ill defined. Therefore, when a dead catalog is detected, a Java error is thrown, which is meant to halt the agent. If, on the other hand, the catalog is still functioning, the error handler checks to see if a working stub for the failed agent exists. A working stub may exist in the catalog if a user has shut an agent down and then restarted it (agent swapping). If there is no working stub, the error handler tries restarting it by issuing a new reliesOn(). When a handle is returned, the problematic method invocation will be repeated, and the error handling process may in fact repeat itself, *e.g.* due to an agent that continually crashes. The current exception handler has no sense of "frustration", *i.e.* it will

[5] A Metaglue catalog lookup() checks the RMI registry for a binding of a stub class file to the provided agent name. Next, the catalog makes a test method invocation into the stub to ensure that the agent is "alive". If the agent is alive, the lookup() will return the stub. If the agent is not found "alive", a RemoteException will be generated by the JVM and caught inside lookup(). The binding will be removed from the RMI registry and the caller notified that the agent is not running.

repeat the described process indefinitely. Nor does the exception handler try delaying between restarts. I have, however, provided the programmer with a means for modifying this behavior. The exception handler can be viewed as a monitor that can be swapped out and dynamically replaced with a user's implementation at run time. Therefore, a user can replace the error handling logic for those agents that he would like, while maintaining the default behavior for the rest of the agents. Furthermore, the exception handler is provided with the name of the method that failed. If a user wanted to have different error handling behavior for different methods, he could accomplish this by checking against the method name. Another modification of the error handler would treat subclasses of RemoteException differently, *i.e.* different communication failures can be handled in different ways.

When trying to introduce the EHA wrappers to the Metaglue system, it was initially felt that the wrappers should be maintained by the Metaglue catalog. When reliesOn() performs a catalog lookup(), it would receive an EHA wrapper instead of the stub. To reliesOn(), as well as the agent, the two are indistinguishable, since agents are always referenced by their interface, and both implement the *interface* file for the agent. When the catalog was implemented with the appropriate logic, it was found that this approach, though attractive in theory, proved to be a poor choice. Calls into the catalog are in fact remote method invocations (they are inter-agent communications). The RMI protocol states that if an object that implements the Remote interface is passed as an argument in a

remote method invocation, that object must have a corresponding stub and skeleton file. Because our wrappers implement the agent interface definition file, and these indirectly extend Remote, the wrapper itself extends Remote. For this approach to work, we would need to create stubs for the wrappers. This is clearly a bad idea, since we are back to our original problem of having unwrapped remote method invocations.

The solution was to return the wrappers from **local** method call. Examination of the system showed that reliesOn() itself is called using local method invocations (it is a primitive inherited from AgentAgent), and so the wrapping logic was placed there.

When reliesOn() is invoked, it checks to see if a wrapper for the agent exists. If it does, it instantiates the agent and returns the wrapped version. If reliesOn() can not find a wrapper, it will return an unwrapped version. Therefore a programmer can choose not to wrap certain agents.

EHA wrappers can not be passed as arguments in remote method invocations for the same reason they could not be returned from the Metaglue catalog. An EHA wrapper implements the Agent interface, and therefore also the Remote interface. To allow the EHA wrappers to be passed between agents, we would need to generate stub files. This brings us back to the original problem of needing to generate stubs for the wrappers. Therefore,

Metaglue insists that the programmer pass references to agents using the AgentID and not the reference to the agent.

With the introduction of the EHA wrappers, Metaglue can now persistently maintain agent connections. If an agent programmer shuts an agent down, the wrapper will detect the failure and attempt to reestablish the connection with no programmer intervention. This fulfills the requirements of the second design goal described in the first chapter, and provides the programmer with a means to introduce traffic-flow control or a debugging system.

4.2 ASYNCHRONOUS RELIESON()

This discussion of asynchronous reliesOn() is detailed and entails substantial modifications to four aspects of the system:

1. reliesOn()
2. The EHA wrappers
3. The Metaglue catalog
4. reliesOnSynch()

4.2.1 RELIESON()

When reliesOn() is called, the first thing it does is try to place a marker in the catalog notifying other agents that the designated agent is being started. This marker is an instance of a class called AgentPlaceHolderAgent that has few methods, but has a rmic generated stub file and can be stored in the RMI registry. At this point the store in the catalog can succeed, meaning that no other thread is trying to start the agent, or it can fail meaning that an instance of the agent is already being initialized, or has already been initialized and the appropriate stub stored in the catalog. If the catalog add fails, reliesOn() returns an instance of the EHA wrapper.

If reliesOn() successfully adds the AgentPlaceHolderAgent to the catalog, it starts a new thread that actually instantiates the agent, and immediately returns an instance of the EHA wrapper. When instantiation completes, the thread replaces the AgentPlaceHolderAgent stored in the catalog with the actual stub to the agent. The AgentPlaceHolderAgent is, as its name implies, simply a placeholder. The placeholder can be stored in the catalog and easily identified.

4.2.2 EHA WRAPPERS

Clearly, the EHA wrappers are a vital aspect of asynchronous agent startup. They provide an object that can be returned immediately without concern for dependencies. When the

EHA wrapper is returned from reliesOn(), it does not have a valid reference to an agent stub. In the sample LampEHA.java code provided in Figure 10, it is clear that the wrappers ensure they have a valid (non-null) value for their interface reference. If the wrapper detects that it does not have a handle to the stub it will invoke a reliesOnSynch() that will block until a handle to the agent is available.

4.2.3 METAGLUE CATALOG

The Metaglue catalog had to be modified so that if an add() was performed, it would allow a rebind of a real stub replacing an AgentPlaceHolderAgent, but would disallow rebinding an AgentPlaceHolderAgent for another AgentPlaceHolderAgent, or for an agent stub. Furthermore, the lookup() method was modified so that it would block if the provided agent name is bound to an AgentPlaceHolderAgent. If there is no binding, lookup() fails.

The AgentPlaceHolderAgent stored in the catalog can belong to a MVM that has crashed or was killed by the agent programmer. Therefore, if someone else tries starting the agent, Metaglue needs to know whether the agent is still being started, or if the agent's JVM has died. When a catalog add()/lookup() is performed, and an AgentPlaceHolderAgent is bound in the RMI registry, a test is performed to ensure that the AgentPlaceHolderAgent is still responding. If the

AgentPlaceHolderAgent is not responding, the add() will succeed (with a rebind) or the lookup() will fail (and not block).

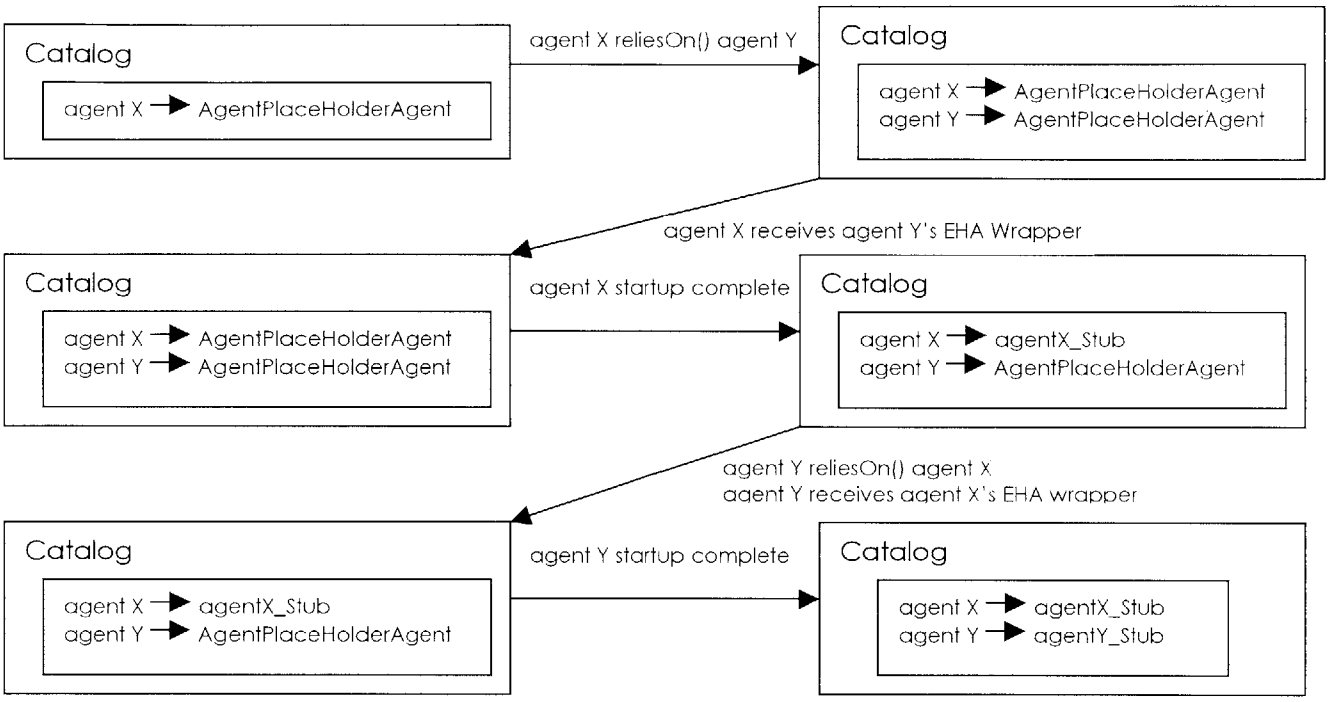
4.2.4 RELIESONSYNCH()

The instantiation of an agent still needs to be performed synchronously. Imagine the asynchronous (independent thread) startup of the agent has failed. At this point, the agent that relied on this agent believes it has a working reference to the target agent, and will perform a method invocation. However, the wrapper detects that it does not have a valid reference. Therefore, the wrapper calls a reliesOnSynch(), that synchronously starts up the agent. This method will not return until it has generated a valid instance of the target agent, either through a catalog lookup or through a new instance. Therefore, if the agent is buggy and always fails during construction, reliesOnSynch() will loop, notifying the user that it is having difficulty starting the agent while continuously attempting the instantiation.

Similar to reliesOn(), reliesOnSynch() also attempts to store an AgentPlaceHolderAgent in the catalog. If it fails (again, because the agent name is bound to either a placeholder or a real stub) it calls the catalog's lookup() which will block if the binding is to an AgentPlaceHolderAgent, or return the agent stub if it's available. If the AgentPlaceHolderAgent store succeeds, reliesOnSynch() will start the agent normally and return the stub when instantiation is complete.

In this new system, what happens if agent X depends on agent Y and agent Y depends on agent X? Assume both agents have been primed and EHA wrappers exist for both. While agent X is starting up, it has a placeholder saved in the catalog. When agent X invokes reliesOn() with agent Y as an argument, the method will immediately return the EHA wrapper to agent Y. Furthermore, agent Y will be started in its own thread. When agent Y declares its dependence on agent X, the reliesOn() will return immediately with an EHA wrapper, regardless of whether agent X has completed the startup or not. At this point, both agents can complete their initialization in good time, and will not block each other. Upon the first method invocation, the method will wait for the stub to be stored in the catalog and then proceed normally.

Figure 15 Asynchronous reliesOn()



Chapter 5

CONCLUSION

Metaglue is being used by seven agent programmers on a regular basis. The programmers report that the system is simple to use, yet powerful enough for the demands of environments like Hal and the Intelligent Room. With only three primitives, new programmers can be trained to use Metaglue quite quickly.

My extensions to Metaglue have served to simplify the programmer's task, as well as make him more productive. Catalog interference is much less common, and programmers need only concern themselves with Metaglue Virtual Machines and not glue spreaders. Programmers use dynamic agent reconnection extensively as they stop and restart agents without needing to restart their applications. Furthermore, the system is more robust and reliable now that components can fail, and Metaglue will automatically restart the agents.

It is interesting to note that none of the agent programmers have replaced the AgentExceptionHandler's default behavior. This serves to demonstrate that agent programmers have found it unnecessary to modify the reconnection behavior, and that the existing protocol is sufficient for most uses.

The possibility of circular agent dependencies has also been well received by the programmers. Various aspects of Hal have been modified to take advantage of this modification. However, asynchronous agent startup has made debugging a more challenging endeavor. Isolating bugs is more difficult when the path of execution is no longer linear through agent initializations.

As reported by the agent programmers, these modifications have proven to be an improvement over the original Metaglue implementation.

5.1 Future Work

There remain areas in Metaglue that can be improved. The current implementation of the EHA wrappers is overly simplistic. These wrappers should include the concept of "frustration". If agent reconnect has failed for a certain number of minutes, or a certain number of attempts, user intervention should be requested. Furthermore, performance could be improved if the wrappers were to delay before reissuing the startup commands.

The addition of load balancing to Metaglue would make the system more powerful. This capability will become increasingly important as applications become more complex and demanding on their hosts.

Finally, Metaglue is still missing an effective debugging environment. Several programmers have attempted to develop a Metaglue debugger, all with limited success. Writing a symbolic debugger for distributed programming environments is a very difficult task. As with load balancing, the need for a symbolic debugger increases with the complexity of applications being developed.

Metaglue remains a system in continuous development. We have found the fundamental design to be sound and easily expandable. With my modifications and extensions, Metaglue has been improved, and is more effective in multi-agent programming environments. There remain, however, areas in need of further development.

Chapter 6

BIBLIOGRAPHY

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

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

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

Horstman, Cay S., Cornell, Gary. *Core Java 1.1: Fundamentals Volume 1*. Prentice Hall Computer Books, 1997.

Horstman, Cay S., Cornell, Gary. *Core Java 1.1 Volume II: Advanced Features*. Prentice Hall Computer Books, 1998.

Horstman, Cay S., Cornell, Gary. *Core Java 2 Volume 1: Fundamentals*. Prentice Hall Computer Books, 1997.

Java Remote Method Invocation Specification. Revision 1.50, October 1998.

Maes, Pattie. "CHI97 Software Agents Tutorial." CHI, April, 1997.

Minsky, Marvin. *The Society of Mind*. Simon & Schuster, Inc. 1988.

Phillips, Brenton. *Metaglue: A Programming Language for Multi-Agent Systems*. Master of Engineering Thesis. 1999.