MCC Technical Report Number ACT-RA-317-90

A DAI Communication Aide

Michael N. Huhns, David Murray Bridgeland, and Natraj Vidur Arni

October 1990

MCC Nonconfidential

Abstract

This report describes a mechanism for communication among computational agents and expert systems of various capabilities. The mechanism is a communication aide that can be attached to an existing expert system, enabling the expert system to interact with the aides of other expert systems and agents. The aide has been designed with the help of a distributed design tool, called Verdi, that helps detect deadlocks. As a result, the aide has been implemented as a pair of communicating processes that interact with the attached expert system and other aides in a deadlock-free manner. Communications occur through a layered protocol and a set of requirements that a knowledge-based system must meet in order to participate in using the protocol. The mechanism has been implemented in the *RAD* distributed expert system shell at MCC.

Microelectronics and Computer Technology Corporation
Artificial Intelligence Laboratory
3500 West Balcones Center Drive
Austin, TX 78759-6509
(512) 338-3718
huhns@mcc.com

1 Motivation

Expert systems need to communicate with each other, to share data and conclusions, and to cooperate in solving problems. Unfortunately, today's expert systems are developed separately, without an eye to future communication. Often, expert systems that could advantageously communicate are not even written in the same reasoning language. This paper is the specification of a *communication aide* designed to allow a heterogeneous group of expert systems to communicate.

One could approach this problem by designing a new expert system language that allows and supports communication. We have taken this approach in the past: we have built an expert system shell in which one can build communicating agents [Arni *et al.* 1990]. However, we believe this approach is insufficient, as many of the expert systems that need to communicate are already written. They are written in an assortment of languages: OPS5, Nexpert Object, KEE, Prolog, etc. Porting an application to a new language—even one that offers new functionality—is expensive and time-consuming. It is better to allow existing systems to communicate as they are written.

This requirement poses three difficult engineering constraints:

1. We do not have control of all the expert system shell software. For example, if we want an OPS5 agent to communicate, we cannot expect to change OPS5 to do this.

2. The expert system languages are all syntactically different and, to some extent, even semantically different. If an OPS5 agent wants to ask a Nexpert Object agent a question, it should not have to know how to phrase the question in the Nexpert Object language.

3. Since we want existing expert systems to communicate, we want to minimize the changes required to these existing expert systems. However, we expect some changes to be necessary. For example, if an existing system is to tell the result of its reasoning to another agent rather than to a user, it must at least know the name of the other agent.

4. We want to allow communications to occur over various types of networks and media.

To meet these constraints, we have refined the concept of a computational agent. In our view, an agent is composed of two parts: a reasoner and a communication aide[1]. They communicate with each other through pipes (UNIX pipes). The reasoner is a knowledge-based system written in an expert system language (e.g., OPS5), and is responsible for all reasoning. The aide manages all communications between its reasoner and other agents.

Since a set of agents working on a problem is a distributed system, we inherit mainstream distributed system issues. In particular, we wish to avoid undesirable situations such as deadlock. To that end, we have used the state-of-the-art visual design tool Verdi[Verdi 1990] to model and simulate a set of agents in action. We describe the modeling and simulation at the end of this document. In the next section we describe an architecture for agents, focusing on the communication aide.

2 Agent Architecture

We first provide a definition for an agent and a coarse categorization of agents into several classes, based on their capabilities. How is a basic agent represented and what can such an agent do? We make several assumptions about the characteristics and capabilities of a computational agent. First, we assume (as in MACE [Gasser *et al.* 1987]) that an agent is an active object with the ability to perceive, reason, and act.

We further assume that an agent has explicitly represented knowledge and a mechanism for operating on or drawing inferences from its knowledge. This precludes from consideration as an agent either a database system (because it has no reasoning mechanism) or a neural network (because its knowledge is not represented explicitly). Third, we assume that an agent has the ability to communicate. This ability is part perception (the receiving of messages) and part action (the sending of messages). In a purely computer-based agent, these may be the agent's only perceptual and acting abilities. It is this communication ability that we wish to generalize and standardize, in order to enable separately developed computational agents to interact. We will then extend this ability in order to allow sub-

[1] In this document "aide" and "communication aide" are used interchangeably.

agents, such as neural networks, databases, and simulation programs, to interact with computational agents (as defined here).

A computational agent is presumed able to perform three basic actions:

1. send a message that represents some knowledge
2. receive a message that represents some knowledge
3. reason, using the knowledge it has available.

We consider both binary and n-ary communication protocols. A binary protocol involves a single sender and a single receiver, whereas an n-ary protocol involves a single sender and multiple receivers. A protocol is specified by a data structure with the following five fields:

1. sender
2. receiver(s)
3. language in the protocol
4. encoding and decoding functions
5. actions to be taken by the receiver(s).

2.1 Aide Architecture

We expect our agents to exist in the environment shown in Figure 1. The agents are fully connected by a local area network, which has a *nameserver* that keeps a list of addresses of all agents. Every agent registers itself with the nameserver. The architecture of an agent is shown in Figure 2.

The aide is implemented as two processes, one that manages the reception of external messages and one that manages the transmission of external messages. As described later, this use of two processes prevents communication deadlocks. Figure 3 shows the architecture of the communication aide.

The aide handles requests from its reasoner, sends messages from the reasoner to other agents on the network via a relay process, receives messages from other agents, and buffers incoming messages from them for the

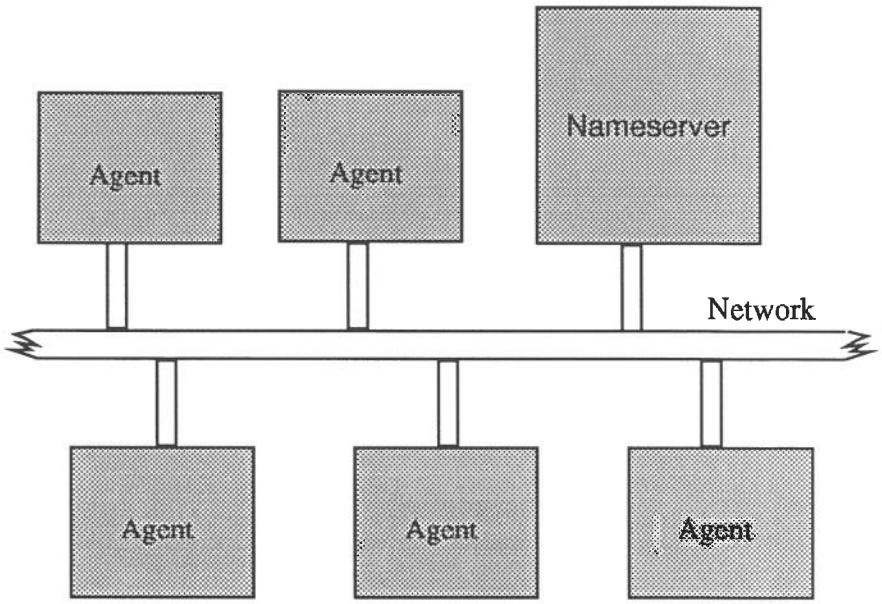

Figure 1: Multiagent system architecture

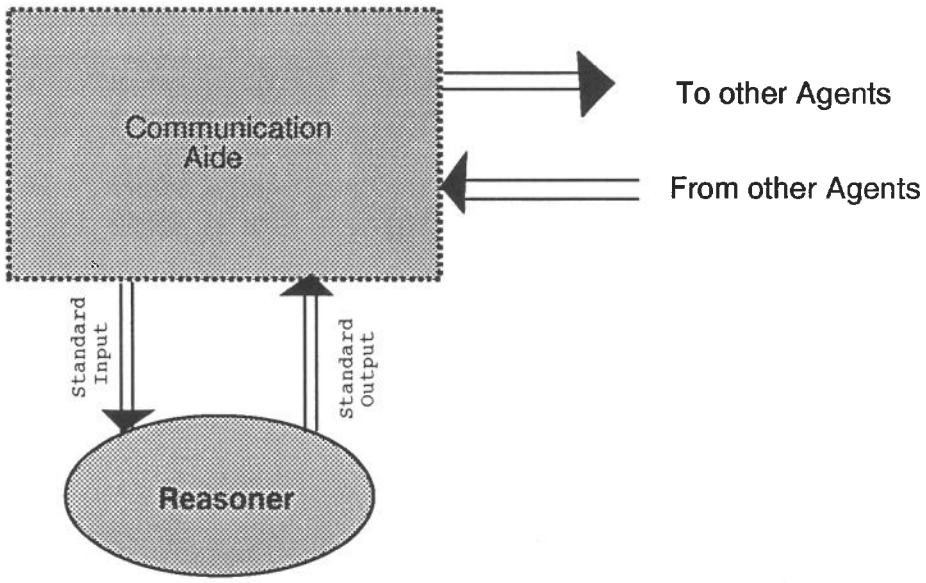

Figure 2: Architecture of an agent

reasoner's later attention. Aides also translate messages sent between reasoners written in different languages. A reasoner communicates with its aide in its native language: e.g., OPS5, KEE, etc. Aides communicate with each other across the network in SURF—SUperset Reasoning Format—which is meant to be a superset of the reasoning languages of existing expert system shells [Genesereth *et al.* 1990]. Hence, an aide must translate from the native language of its reasoner to SURF, and from SURF back to its reasoner's native language. This document specifies only the communication aspects of an aide. The syntax and semantics of the SURF language, and the details of translation between various native expert system languages and SURF, are described elsewhere.

The aide operates in either *normal mode* or *wait mode*. In normal mode, the aide listens simultaneously for messages from the reasoner and from other aides. When a message is received, appropriate actions are taken, depending upon the source of the message and its type[2]. Noncommand-type messages from other aides are buffered in the aide's *message buffer*, whereas messages of type *Command* are executed by the aide. In *wait mode*,

[2]Message types are discussed in a later section.

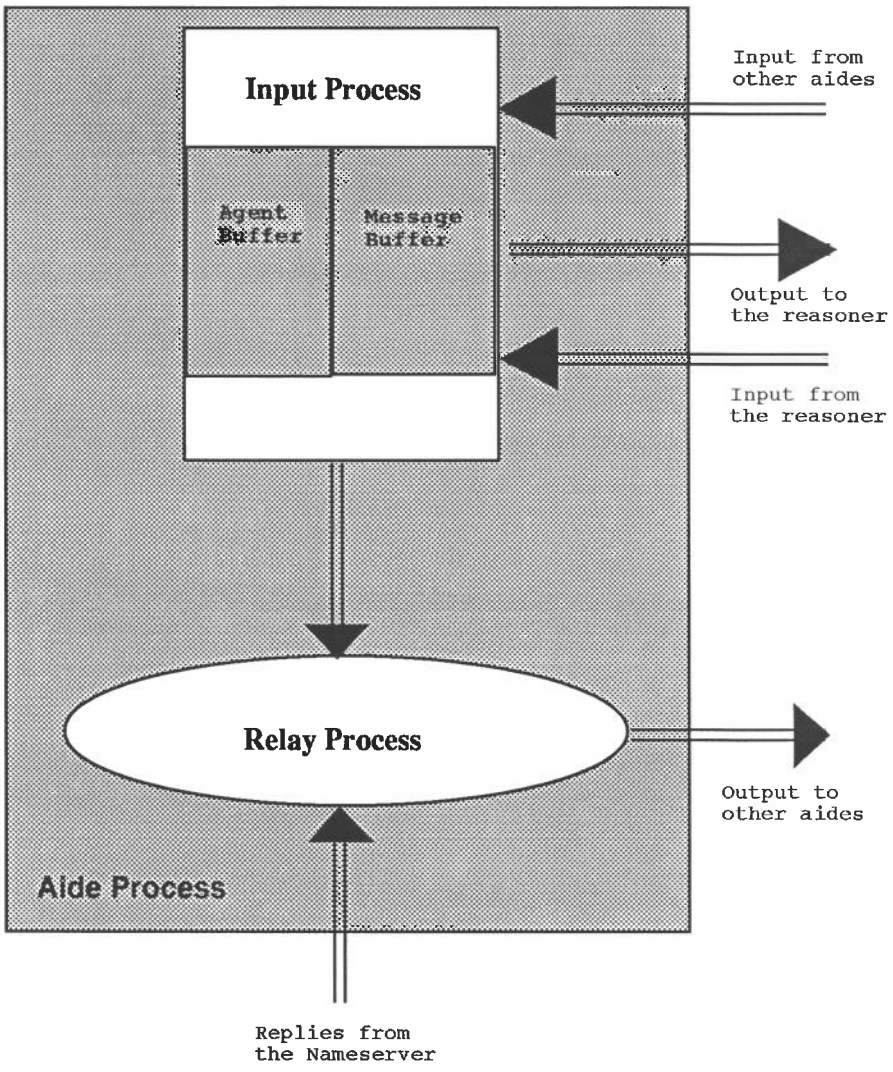

Figure 3: Architecture of a communication aide

the aide listens for messages from other aides only. It does not listen to the reasoner. The reason for this behavior is explained later in a discussion of message communications. In addition to the message buffer, the aide also has a list of known agent addresses, built by querying the nameserver. It is from this list that it finds the address of a given agent in the Internet domain.

2.2 Relay Process

A communication aide uses a *relay process* to send a message to another agent on the network. As shown below, the relay process is necessary to prevent deadlocks. The relay process reads messages from the aide and transfers those messages to the intended recipient agent on the network. The relay process also has to interact with the nameserver, in cases where the recipient agent has changed its address. These interactions are described in a later section of this document devoted to interactions with the nameserver.

We require a separate relay process to send messages, because an aide A_i, wanting to send a message to another aide A_j $(i \neq j)$, has to wait for A_j to be ready to receive the message. If, at that instant, aide A_j is trying to send a message to aide A_i, a deadlock occurs, since both aides are waiting for each other to be ready to receive. With a relay, the aide transfers the message through a pipe to the relay for transmission, while it is free to listen for incoming messages from the reasoner and/or the outside world.

Even in this scenario there is potential for a deadlock. As mentioned above, communication between an aide and its relay is done using a pipe. In most implementations of UNIX, the pipe has a finite length, say S_p.[3] If the size of the message sent from the aide to its relay is greater than S_p, the aide **blocks** till enough space is freed up in the pipe by the relay[4] so that the aide can finish writing to the pipe and resume waiting for external messages. Consider the situation shown in Figure 4, where messages M_1, M_2, M_3, and M_4 are all larger than S_p. Assume that aide A_i wanted to send M_1 and M_2 to aide A_j $(i \neq j)$. Aide A_j, on the other hand, is trying to send messages M_3 and M_4 to A_i. Relay R_i receives M_1 and tries to send it to A_j,

[3] The length of the pipe is 4K bytes for SunOS 4.0.

[4] The relay frees up the pipe buffer by reading from it.

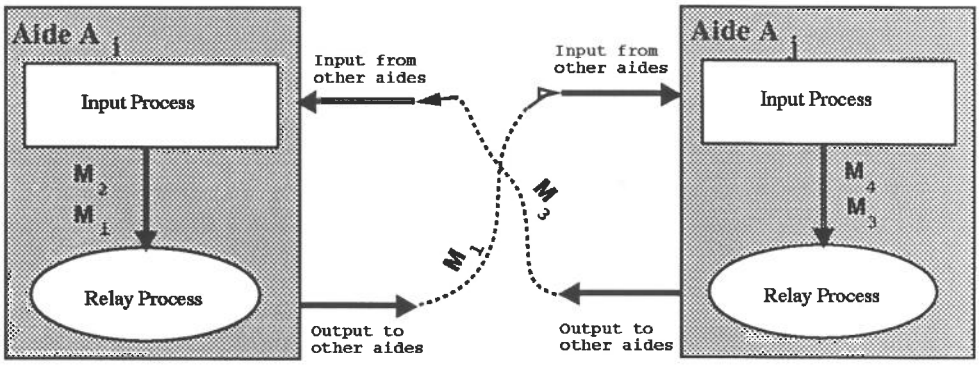

Figure 4: Communication deadlock among aides

whereas relay R_j receives M_3 and tries to send it to A_i. But aides A_i and A_j are not yet ready to receive messages from relays R_j and R_i, respectively. This could cause R_i and R_j to block. Aides A_i and A_j, respectively trying to "pipe" M_2 and M_4 to R_i and R_j, could block too, since the message sizes are greater than the pipe length. This would lead to a deadlock.

We resolve this deadlock using the following strategy. A relay R_i reads its input from a pipe attached to its controlling aide, A_i. After reading a complete message, the relay **forks** a copy of itself, resulting in two identical processes called **parent** and **child**. The child process takes care of the transmission of the message to a recipient aide and then dies, whereas the parent returns to seek input from the aide.

It should be noted that the relays were introduced in the first place to avoid the kind of deadlock mentioned above. By using the relays, we had just pushed the problem one step away—not really solved it. Forking another process solves the problem. It could be argued that we do not really need a relay process to do our transmissions, since we can fork the aide process to do this. That would work, but it would be computationally expensive, since aide processes are much larger than relay processes.

3 Interagent Communications

Our goal is to provide a means by which agents of different capabilities can communicate. The communications must therefore be defined at several levels, with communication at the lowest level used for communication with the least capable agent. In order to be of interest to each other, the agents must be able to participate in a dialogue. Their role in this dialogue may be either active, passive, or both. This allows each to function as a master, slave, or peer, respectively.

In keeping with the above definition for and assumptions about an agent, we assume that an agent can send and receive messages through a communication network. The communication proceeds through several phases, viz. Reasoner to Aide, Aide to Reasoner, Aide to Relay, Aide to Aide, Relay to Nameserver, and Aide to Nameserver. Messages passed between these various entities use distinct formats, according to the *type* of the message. The messages can be of several types, as defined next.

3.1 Message Types

There are two basic message types: assertions and queries. Every agent, whether active or passive, must have the ability to accept information. In its simplest form, this information is communicated to the agent from an external source by means of an assertion. In order to assume a passive role in a dialog, an agent must additionally be able to answer questions, i.e., it must be able to 1) accept a query from an external source and 2) send a reply to the source by making an assertion. Note that from the standpoint of the communication network, there is no distinction between an unsolicited assertion and an assertion made in reply to a query.

In order to assume an active role in a dialog, an agent must be able to issue queries and make assertions. With these capabilities, the agent then can potentially control another agent by causing it to respond to the query or to accept the information asserted. This means of control can be extended to the control of subagents, such as neural networks and databases.

An agent functioning as a peer with another agent can assume both active and passive roles in a dialog. It must be able to make and accept both assertions and queries.

The message types fit into a (single) inheritance hierarchy, with some types as *subtypes* of others, as shown in Figure 5. Only a few of the high level message types are presented here. Other message types, derived from work on speech-act theory [Allen 1987] and listed in Table 1, will be developed over time. The lower three types in this table are used to implement a Contract-Net Protocol [Smith 1980].

Message types are encoded as bit vectors, with each type having a unique index into the vector, and with a given type's bit vector having 1s in all the indices of which the type is a subtype, and 0s elsewhere. A previously unseen type can be classified by walking down a tree of known types, comparing the unknown type with the known ones by bit-anding and comparing the result with the known type. The bit vectors are 32 bits long, allowing 32 different message types to be transmitted. It is essential that the aide provide support for this type system, so that an incoming message of a type unknown to the aide can be recognized as a subtype of a type the aide knows how to handle.

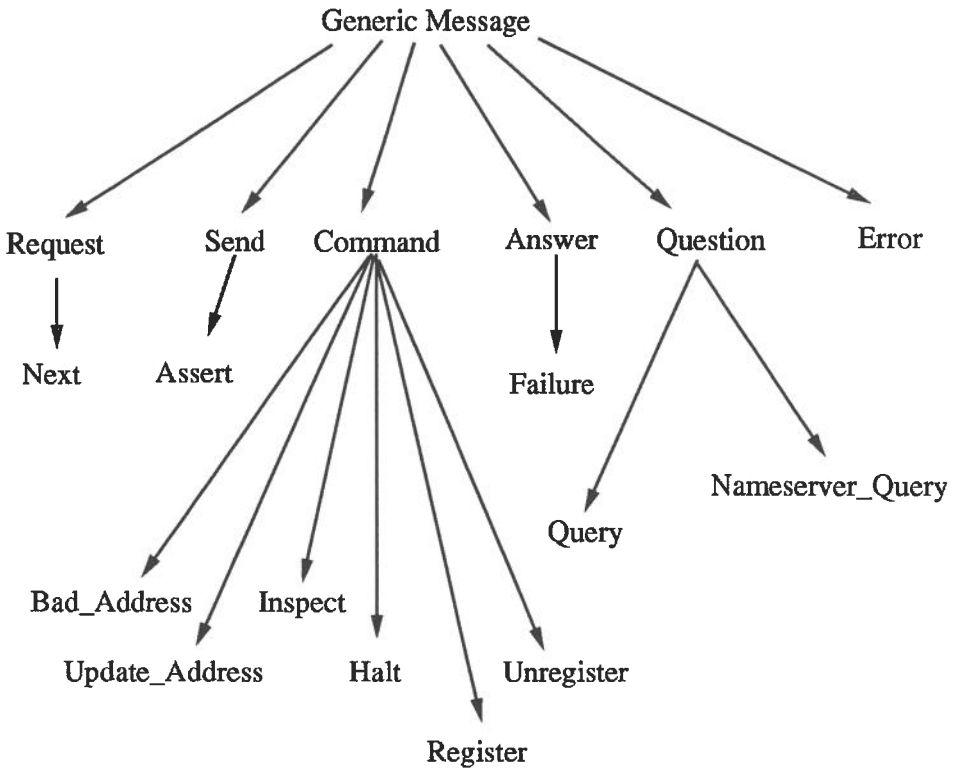

Figure 5: Hierarchy of interagent message types; a link represents a subtype relationship

Table 1: Interagent Message Types

Communicative Action	Illocutionary Force	Expected Response
Assertion	Inform	Acceptance
Retraction	Inform	Acceptance
Acceptance	Inform	none
Query	Question	Reply
Reply	Inform	Acceptance
Request	Request	Acceptance/Refusal
Refusal	Inform	Acceptance
Command	Request	Acceptance
Permission	Inform	Acceptance
Explanation	Inform	Acceptance
RFP	Request	Proposal
Proposal	Inform	Contract/Refusal
Contract	Inform	Acceptance

3.2 Communication between a Reasoner and its Aide

The reasoner and its aide are separate processes. The reasoner is just an expert system shell process, writing to the standard output stream and reading from the standard input stream as if these streams were connected to a monitor and a keyboard. Instead of a monitor and a keyboard, however, the aide reads the reasoner's standard output stream and writes to the reasoner's standard input stream. The aide and reasoner communicate in this manner so that no change is required in the expert system shell running the reasoner. This communication method also has the nice property of simplifying the task of debugging an implementation of the aide and reasoners. The aide can be debugged by attaching it to a dummy reasoner process actually connected to a shell, and a reasoner can be debugged by the developer pretending to be an aide connecting the reasoner to the outside world.

Every species of reasoner has its own syntax, e.g., OPS5 commands look very different from Nexpert Object commands. The message type system hides some of this complexity. For example, the OPS5 prompt ">" is interpreted by its aide as a NEXT message. Nexpert Object has a different prompt representing the same message.

There is a standard message syntax between reasoner and aide. It is not expected that any reasoner actually emits or reads messages in the standard syntax. This standard syntax is for building interface and test "reasoners." Further, it is expected than any message from reasoner to aide could in principle be translated (e.g., by using lex and yacc) to the standard syntax and read by the aide in that manner, and likewise for any message from the aide to a reasoner. It may even be useful (for simplicity and modularity) to implement aides for some shell languages in this manner.

Communication between a reasoner and its aide is always initiated by the reasoner: the aide does not send information to the reasoner except when the reasoner has asked. The reasoner can be in either of two states: 1) reasoning, or 2) waiting for the aide. When it is waiting for the aide, the reasoner has always made some request of the aide and is waiting for the fulfillment of that request. While it is waiting, the reasoner cannot communicate with the aide at all, so none of the messages described below are legal. While the reasoner is reasoning, any of the messages are legal, except that it cannot answer a question if no question had been posed (as explained below).

3.3 Messages from a Reasoner to its Aide

The standard syntax for formatting a message consists of the message type, followed by a single space, and then followed by the message arguments (if any) separated by single spaces. The following table summarizes how various entities are formatted in the standard syntax:

message type — an (unsigned) integer representing a (32 bit) bit vector.

typepred — an integer representing a bit vector.

agentname — a series of no longer than 31 printable ASCII characters without spaces.

timeout — an integer greater than -2 and less than 2^{15}. (-1 indicates an indefinite wait.)

msg — a series of printable ASCII characters, indefinitely long and terminated by a null character.

Each of the following message types includes the name of the type, its arguments (if any), and its bit vector in the standard syntax. Each bit vector is written as a string of 1s and 0s, for easier reading.

Request *typepred* 00000000000000000000000000000011

When a reasoner is ready for more input from the outside world, it requests a message from the aide. The *typepred* argument is used to specify what sort of message is acceptable. Any message type that is a subtype of *typepred* is acceptable. If an acceptable message has already been received, the aide should transmit it to the reasoner. Otherwise, the aide should wait indefinitely for an acceptable message to be received from the outside world. If the message received by the reasoner from its aide happens to be of type *question*, the aide then keeps a note of the message id of this question. The aide uses this message id as the id of the answer from the reasoner. (The specification language used for *typepred* may evolve as new uses are found.)

Next 00000000000000000000000010000011

Next is a submessage of **request**. When a reasoner wants a message from the outside world, it sends a **next** request and the aide returns the oldest message. The reasoner waits until a message is returned.

Send *agentname msg* 00000000000000000000000000000101

Send is used to send a nonrequested message to another agent when the sending agent does not expect a response. Once the message has been transmitted to its aide, the reasoner continues its work.

Question *agentname timeout msg* 00000000000000000000000000001001

To send a message that requires a response to another agent, a **question** message is used. Once transmitted to the aide, the reasoner waits until the aide responds. If a timeout is given, the aide will respond before *timeout* time (in number of seconds) has expired. (Note that **Question** is the general message type that is used to support queries of one agent by another, queries of the nameserver, and queries of a person by an agent.)

An agent cannot send a question to itself. Such a message is caught and discarded by the aide, and some appropriate response (dependent on the species of reasoner) is returned to the reasoner.

Answer *msg* 00000000000000000000000000010001

When a reasoner is asked a question by some agent, its response is relayed to the aide by means of an **answer** message. This response is then sent to the questioning agent.

At any time the reasoner either has a pending question from some agent, or it does not. In the former case, the aide keeps track of who had asked the question, and thus the reasoner need not identify the address to which an answer is sent. In the latter case, an **answer** message from the reasoner to the aide is illegal, and is discarded by the aide.

3.4 Messages from an Aide to its Reasoner

The aide also sends information to the reasoner, in reply to a message of type **request**. As with messages from the reasoner, there is a standard syntax, and this syntax is very similar to that for messages going the other direction. In particular, all messages start with the message type, and the same types are used as with analogous messages from the reasoner to the aide. The message type is followed by a space and then the arguments of the message, with the same encoding as described before.

Send *agentname msg* 00000000000000000000000000000101

The agent *agentname* has sent this agent a message.

Question *agentname msg* 00000000000000000000000000001001

The *agentname* has sent this agent a question. The reasoner is not expected to make another request until this question is answered.

Answer *msg* 00000000000000000000000000010001

This is the response to a question that the reasoner has asked. The reasoner should keep track of the agent it has queried. The aide also makes sure that this *answer* is the one for which the reasoner has been waiting. This is done by remembering the message id of the question and the target agent name when the question was sent out, and comparing them with the message id number and the sending agent name when an answer is received.

3.5 Communication between an Aide and its Relay

The aide uses a relay process to communicate with other aides on the network. Conceptually, this relay process is very simple: it just reads a message from the pipe connecting it to the aide and relays that message to

its destination. After it finishes that communication, it waits for the next message from the aide.

The messages from the aide to the relay include (in order, and separated from each other by single spaces) the agent name, the destination host, the destination port, and an arbitrarily long message, terminated by a null character. Although the message is divided into other fields from the aide's point of view, the relay sees it as a single entity.

As described before, the relay at this point forks into two processes, parent and child. The parent waits on the aide for its next message to relay, whereas the child process attempts to transmit the message. If the child relay is unable to connect to the host and port specified, it contacts the nameserver[5] to determine a new host and port. If the nameserver has a new host and port, the child relay attempts the new address. On failure, the child relay continues to update the address via the nameserver as long as the nameserver continues to return different addresses for the agent. If the nameserver ever returns the same address, the child relay abandons its attempt to send this message, writes an appropriate error to its standard error stream, and reports to the aide that the address of the given agent is bad by using a **bad-address** message to the aide via the aide's public port.

If the child relay succeeds in relaying the message, but only with a new address for the destination agent, it must communicate this information back to the aide, so the aide can update its local database. It does this by sending an **update-address** message to the aide via the aide's public port. This message type is described in the next section, with the rest of the public port messages.

3.6 Communication among Aides

Aides send messages to other aides according to a type hierarchy of messages. By aide-to-aide communications, we mean relay-to-aide communications, since aides use relays for transmissions. The type hierarchy overlaps the type hierarchy for communication between the aide and reasoner, so no translation is necessary.

The formatting of aide-to-aide communication is simpler than that of

[5]The relay process already knows the host and port of the nameserver.

reasoner-to-aide, as there is no need to support a variety of different syntaxes. There is a fixed, standard syntax for all messages. All aide-to-aide or relay-to-aide messages have the following fields:

sending-agent — a series of no longer than 31 printable ASCII characters without spaces.

sending-host — the name of the host sending the message. This follows the rules for host names in UNIX.

sending-port — the port (an integer) of the agent sending the message.

destination-agent — a series of no longer than 31 printable ASCII characters without spaces.

message type — an (unsigned) integer representing a (32 bit) bit vector.

id — an integer greater than -1 and less than 2^{15}. Together with the sending agent name, this forms the unique id of a message. References to this message will be in terms of this id.

msg — a series of printable ASCII characters, indefinitely long and terminated by a null character.

Following are the message type codes and their arguments. An aide receiving these messages has to parse them accordingly.

Send *msg* 00000000000000000000000000000101

Msg is sent to the destination aide and buffered.

Question *msg* 00000000000000000000000000001001

Msg is a question sent to the destination agent. The corresponding message tag is remembered. The sending agent is expecting a reply. In this mode, the sending aide goes into the WAIT mode until it gets a reply or it "times out." In the latter case, it sends a failure to its reasoner.

Answer *msg* 00000000000000000000000000010001

Msg is an answer to a question originally sent by the destination agent. If the destination aide were still waiting for an answer, then it would compare the id of this message with the message id of the question it sent. If they are the same, *Msg* would be the answer to its query, else this *answer* would be discarded.

Command *msg* 00000000000000000000000100001

Msg is a command for the aide of the destination agent. Commands only affect the aide and are invisible to the reasoner.

Inspect 00000000000000000100000100001

The destination aide is asked to dump its entire state in a human readable form to the standard error stream. This message type is useful for debugging.

Bad-address *agent* 00000000000000000010000100001

The destination aide is advised to delete the address entry of *agent* in its agent buffer. *Agent* is a string of no longer than 31 printable ASCII characters without spaces.

Update-address *agent host port* 00000000000000000100000100001

The destination aide is advised to update its address of *agent* to be *host* and *port*. *Agent* is as described above, *host* is a UNIX host name, and *port* is an integer.

3.7 Startup and Shutdown

When an aide is started, it must follow a complicated protocol to ensure that the entire agent starts its operation as a coherent whole. First, the aide requests a port on which to listen. Then it attempts to register the agent's name and the port it just obtained with the nameserver. On failure here, it simply shuts down with an appropriate message to the standard error stream. After successfully contacting the nameserver, the aide attempts to start up a relay process, which knows the host and port of the nameserver, and the reasoner process. On failure to do either of these, the aide deregisters with the nameserver and terminates. Finally, if the aide is successful, it starts listening at its port, and normal operation begins.

4 *RAD* Agents Interacting with OPS5 Agents

The extent or richness of the interactions between a *RAD* agent and an OPS5 agent is determined and can be specified by two independent dimensions. The first is the degree of autonomy of the OPS5 agent: it can be passive and function as a slave to a *RAD* agent, or it can be active and

Table 2: *RAD* Agent Using OPS5 Agent

RAD Command	Message Type	Direction	OPS5 Translation
tell < *pattern* >	statement	$\Longrightarrow$	make
tell < *subclass* >	statement	$\Longrightarrow$	literalize
query	demand	$\Longrightarrow$	ppwm
RAD data	response	$\Longleftarrow$	working memory elements
?	command	$\Longrightarrow$	remove
?	command	$\Longrightarrow$	run

function as a peer to a *RAD* agent. The second is the capability of the *RAD* agent: it can function as a *user* of an OPS5 agent, or it can function as a *developer* of an OPS5 agent. Figure 6 illustrates these relationships.

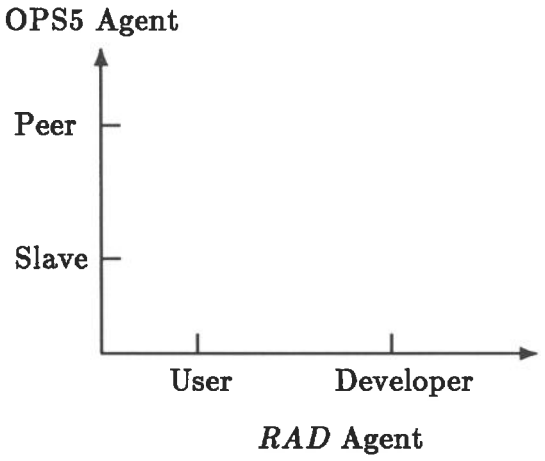

Figure 6: The possible interactions between *RAD* and OPS5 agents

Only the most basic communication capabilities are required for a *RAD* agent to control an OPS5 agent as a user would. In this case, the OPS5 agent does not initiate any interactions between the agents, but simply responds to commands and requests from the *RAD* agent. The interaction language is shown in Table 2.

Additional capabilities are required of the interaction language if an OPS5 agent is to be able to function autonomously as a peer to a *RAD*

Table 3: *RAD* Agent Interacting with Peer OPS5 Agent

RAD Translation	Message Type	Direction	OPS5 Command
tell	statement	$\Longleftarrow$	write
query	demand	$\Longleftarrow$	write, accept
RAD data	response	$\Longrightarrow$	working memory elements

Table 4: *RAD* Agent Developing OPS5 Agent

RAD Command	Message Type	Direction	OPS5 Translation
?	command	$\Longrightarrow$	openfile
?	command	$\Longrightarrow$	closefile
?	command	$\Longrightarrow$	default
?	command	$\Longrightarrow$	strategy
?	command	$\Longrightarrow$	back
?	command	$\Longrightarrow$	excise
?	command	$\Longrightarrow$	wm
?	response	$\Longleftarrow$	working memory elements
?	command	$\Longrightarrow$	pm
?	response	$\Longleftarrow$	production memory elements
?	command	$\Longrightarrow$	cs
?	response	$\Longleftarrow$	conflict set

agent. The OPS5 agent must be able to initiate interactions, as shown in Table 3.

Further capabilities are also required of the interaction language if the *RAD* agent is to act as a developer of an OPS5 agent. In this case, the *RAD* agent must be able to transmit commands that inspect the knowledge, and monitor and modify the reasoning process of the OPS5 agent. These additional commands are shown in Table 4.

The premises of our implementation are that 1) the underlying language should not be modified, and 2) existing systems written in that language should be changed as little as possible. Also, for ease of implementation, a *RAD* agent should emulate a user, rather than a developer, of an OPS5 agent.

5 Implementation Details

The communication aide has been implemented in C++ on a Sun SPARC station running SunOS 4.01. For communication, we use UNIX sockets in the Internet domain. Agent, Aide, and the Relay are modeled as classes. Each of these entities has its own parser to parse incoming messages. Other data structures to perform message buffering and agent look up tables are implemented as C++ classes also. The Aide spawns the relay and the reasoner processes using the *execvp* call in UNIX. Communication between them occurs through standard pipes. To test our system, we used a very simple reasoner interface, where the human being was the real reasoner.

6 A Verdi Model

A collection of agents communicating via aides is a distributed system. Distributed systems are difficult to design, because they are prone to *deadlock*, a state in which several processes in communication are all waiting for something from each other before any can proceed. The reasons for waiting are many: different distributed systems share different resources, and it is contention for these resources that generates different waiting relationships. There are thus several types of deadlocks. Research on the Hecodes distributed expert system [Bell and Zhang 1990] concentrated on deadlocks at the task level, where expert systems had to wait for others to solve subtasks before they could proceed. Hecodes avoided these deadlocks by using a central control system to assign and schedule subtasks.

Our research has focused on preventing deadlocks at the message level, where deadlocks are caused by the protocol and resources of communication among agents. The avoidance of deadlocks is more difficult in our system than in Hecodes, because our agents are autonomous: control, as well as reasoning, is distributed. There is no omniscient central controller to resolve deadlocks.

We thus used the distributed system design tool Verdi^{TM} to help us in designing the communication aide to avoid deadlock problems. Verdi is a visual programming language and simulator for the design of distributed systems. Once a distributed system is designed with Verdi, it can be simulated, with the results of the simulation animating a colorful display of the

system in action. Such an animation is useful for two reasons:

- It reveals how and when deadlocks occur, which Verdi recognizes also.
- It makes it easy to acquire an intuitive understanding of the behavior of the system.

Verdi helped us to find three problems with our original design, each of which would have resulted in deadlock.

6.1 A Simple Model

Figure 7 shows the first of the Verdi models we created. For simplicity at this stage, we modeled only one agent, and modeled the collections of other agents on the network as a single process called "world." The agent itself is modeled by two processes, the reasoner and the aide. The aide communicates to the world and to the reasoner via "interactions," shown in the figure as labeled boxes with double vertical bars.[6] An interaction is *between* two or more processes, and can only happen if all of the processes in the interaction are ready to interact. If more than one interaction is enabled, one is chosen arbitrarily. Interactions are a theoretical construct that we use to model various actual events between processes, including message passing along a network (i.e., between the aide and the world) and pipelines between two processes on the same host (i.e., between the aide and the reasoner).

The behavior of the world is very simple. After an arbitrary delay, the world either sends a message to the agent, or receives a message from the agent. A message is either a *send*, a *question*, or an *answer*. Since each message can either be outgoing or incoming, there are six possible message types. The world will only answer a question (*a_answer_in*) if there is at least one outstanding question. This is implemented by a guard on the *a_answer_in* interaction that prevents that interaction from being ready if there are no outstanding questions. After a message has been sent or received, the world restarts, consisting of an arbitrary delay followed by some message.

[6]The formal interpretation of Verdi diagrams is beyond the scope of this paper[Verdi 1990] but it is expected that the reader can follow the rather informal description given here by referring to the appropriate Verdi diagram.

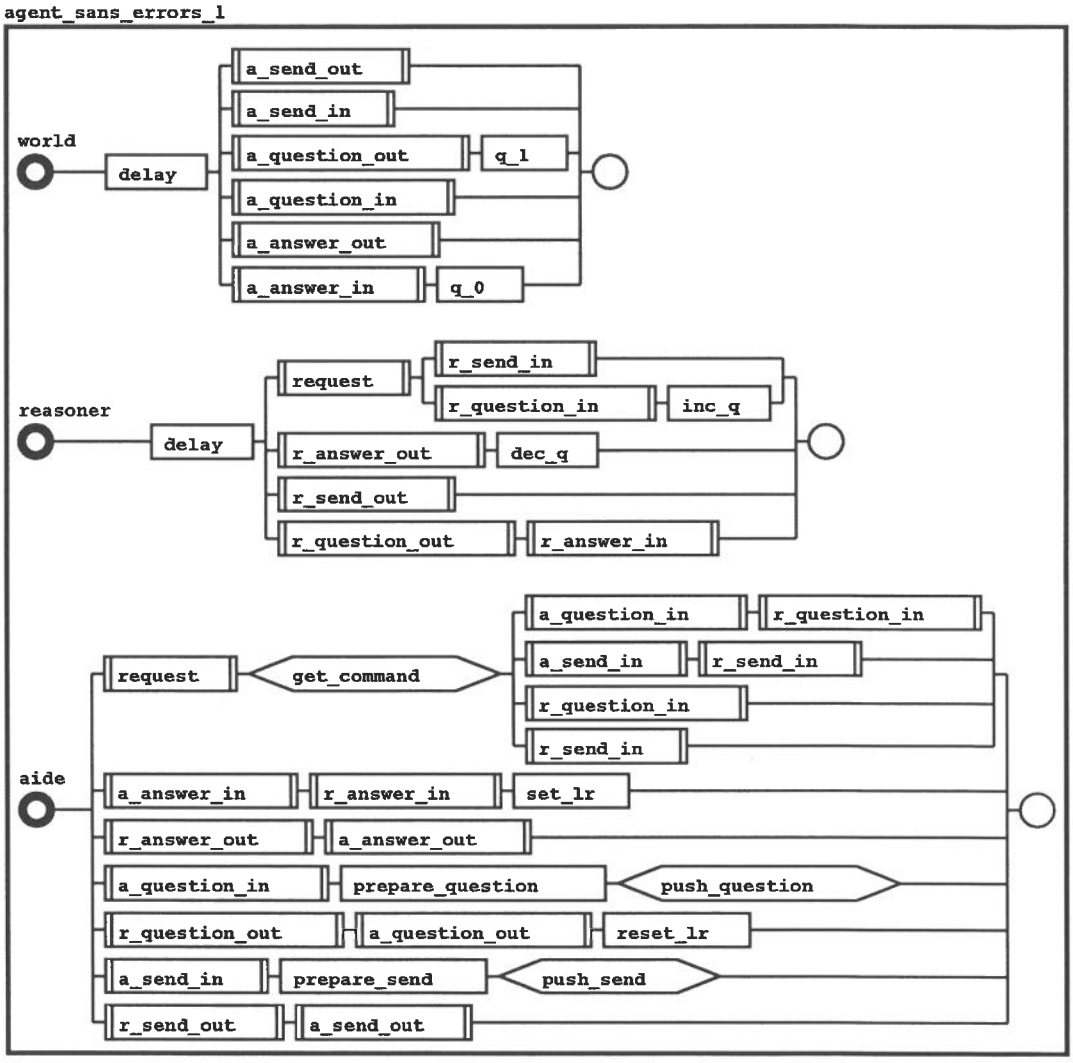

Figure 7: A model of a simple agent

The behavior of the reasoner is only a little more complicated. Initially, the reasoner is reasoning, which is modeled by an arbitrary delay. After this delay, the reasoner answers the questions it has been reasoning about, or listens for the next request from another agent, or sends a message to some agent, or asks a question of another agent. If it asks a question, it waits for an answer. Otherwise the reasoner restarts, by reasoning and then communicating. There are two restrictions on this behavior: the reasoner will not answer a question unless it has been asked one, and the reasoner will not issue a request if there is a pending question. Both of these restrictions are implemented by guarding interactions with appropriate conditions.

The aide joins the reasoner to the world. Initially, it listens for messages from the world (*a_answer_in*, *a_question_in*, or *a_send_in*) or from the reasoner (*r_answer_out*, *r_question_out*, or *r_send_out*), or a request from the reasoner to send the next message from the world. If it receives an answer from the world to a question originally asked by the reasoner, it knows the reasoner must be hanging, waiting on the answer to that question, and so the aide immediately passes the answer to the reasoner and continues. Likewise, if the reasoner wants to answer a question posed by another agent, or send a message to another agent, or ask a question of another agent, the aide passes the message to the world without further ado. If the world wants to query or send a message to the reasoner, the aide cannot immediately pass the message on as the reasoner may not be ready, and cannot wait for the reasoner to be ready as deadlock could occur. Instead, it must queue the message and continue listening. The message is passed on if and when the reasoner asks for it via the request interaction.

When the reasoner requests the next message from the aide, the aide checks its message queue. If empty, the aide waits for the next question or send from the world (*a_question_in* or *a_send_in*). (The reasoner waits as well.) When a message is received from the world, the aide passes it to the reasoner and restarts. If the message queue is not empty, the aide pops the next message off the queue and passes it to the reasoner.

6.2 Modeling Multiple Agents

The approach of the simple model—modeling one agent in detail and the rest as "the world"—is too abstract to exhibit pathological distributed-system behavior. In fact, although we learned much about the process of

modeling in Verdi from building that simple model, we did not refine the design of the communication aide at all as a result of the modeling and simulation. The next step was to build a multiple agent model, shown in Figure 8. There are *agent_count* number of agents. Each agent has an aide, a reasoner, and a relay process. The three processes shown in the diagram are thus *replicated*, once for every agent (and that is what the i : *agent_count* notation means). Further, the names of the interactions are subscripted with is and js to associate them with agents.

At all times, a reasoner is in one of three states. It is reasoning in order to answer a question posed to it by another agent, or it is reasoning without a open question, or it is idle, waiting for the next question or message from the world. This state is modeled by a state variable, which unfortunately is invisible in the visual Verdi representation.

Initially each reasoner is reasoning, without an open question from another agent. From this state, the reasoner may either send a message to another agent, send a question to some other agent, or become idle and request the next message from the outside world. All of these actions are rather similar to their counterparts in the reasoner of the simpler model in Figure 7, but for questioning another agent or sending a message to another agent, the reasoner must decide with which agent to communicate. In the Verdi model, this is decided randomly, and passed to the aide in the second part of a two part protocol (e.g., in the *r_send_out_b_i* or *r_question_out_b_i* interactions). [7] If a request for a new message is issued, and a question is received, the reasoner enters a state in which it is reasoning to answer a question. In this state, it can answer the question, but it cannot request another message until the question is answered. When it does answer the question, the reasoner enters the third possible state, idleness. While idle, the reasoner can do nothing but request the next message from the aide.

The aide is also much the same as the one in the simpler model. Here an aide is actually communicating with another aide rather than "the world," so send, question, and answer interactions within each aide are replicated. There is thus one contact interaction for each other aide. (Again the replication is represented by the j : *agent_count* notation.)

A deadlock was discovered by simulating an earlier multiagent model.

[7] This two stage protocol is not actually required in the system modeled, only in the Verdi model.

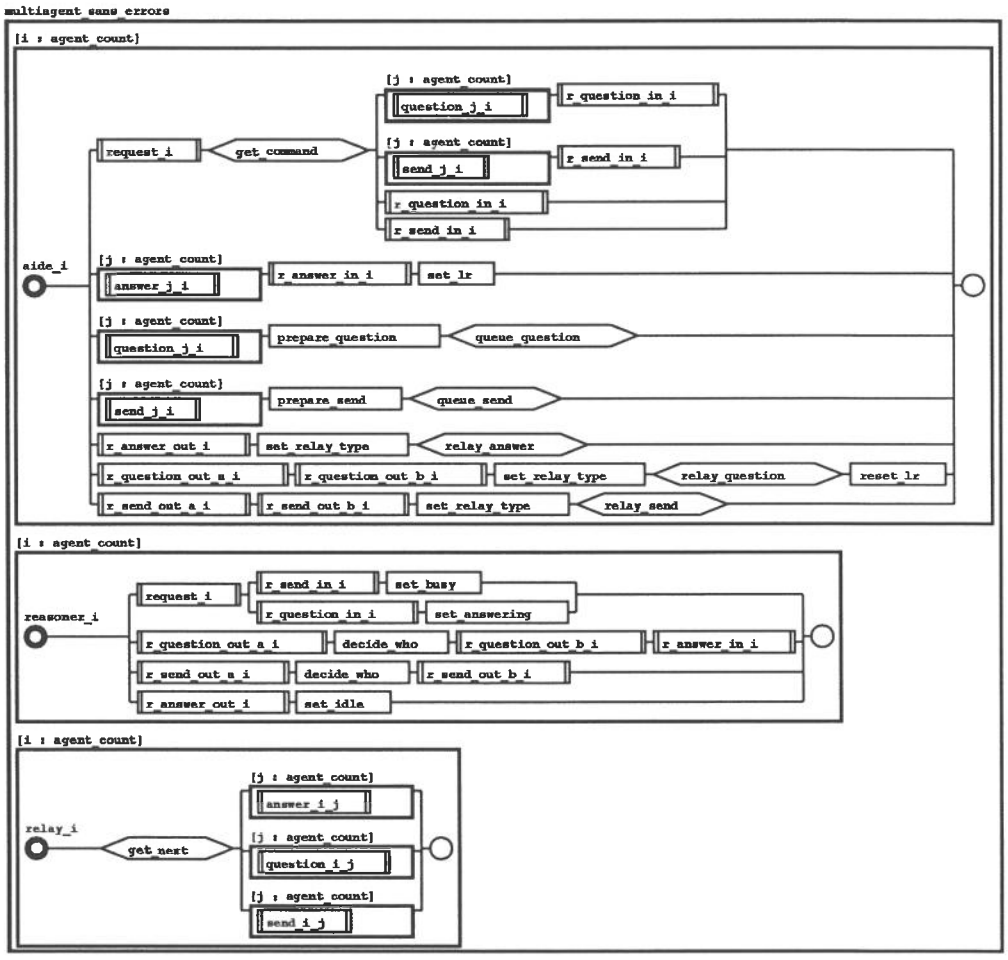

Figure 8: A model of one agent in a multiagent system

This deadlock involved two aides attempting to communicate with each other simultaneously. For example, while the first was attempting to send a message to the second, the second was attempting to query the first. Deadlock resulted because each would wait forever for the other to listen. This deadlock was avoided by the addition of a relay process for communication between two aides. An aide sends a message to another via the relay process. The relay process waits until the listening aide is ready to communicate, and thus the aide itself will never wait for another aide to be listening. The aide talks to the relay process via a UNIX pipe, modeled in Verdi as a queue. The actions *relay_answer*, *relay_question*, and *relay_send* push things on the queue, and the action *get_next* in the relay process pops something off of it.

The relay process itself is simple. It attempts to pop something off the real queue (i.e., it listens to the UNIX pipe) and hangs if there is nothing in the queue. When it is successful, it determines which is the destination agent and what type of message (question, answer, or send) is involved. Finally, it communicates with the appropriate aide using the appropriate message type.

7 Modeling Errors

The aide is built to handle errors, including recognizing and rejecting bogus messages, and a facility to allow questions to time out and be aborted. A Verdi model of a multiagent system with error handling is shown in Figure 9. Associated with each agent is another process for timing out during an unanswered question[8]. The aide interacts with this timeout process by setting the clock, timing out and canceling the question if the aide has waited too long, or canceling the timeout if the answer arrives in time. The aide is also capable of simply rejecting messages. In the system being modeled, this rejection would have some reason. For example, the aide for an OPS5 agent might reject a backward rule sent to it. In the Verdi model, the rejection is random; the aide simply and arbitrarily rejects some of the messages.

[8]This extra process is strictly a modeling artifact; in the modeled system, timeout would be accomplished inside the aide process.

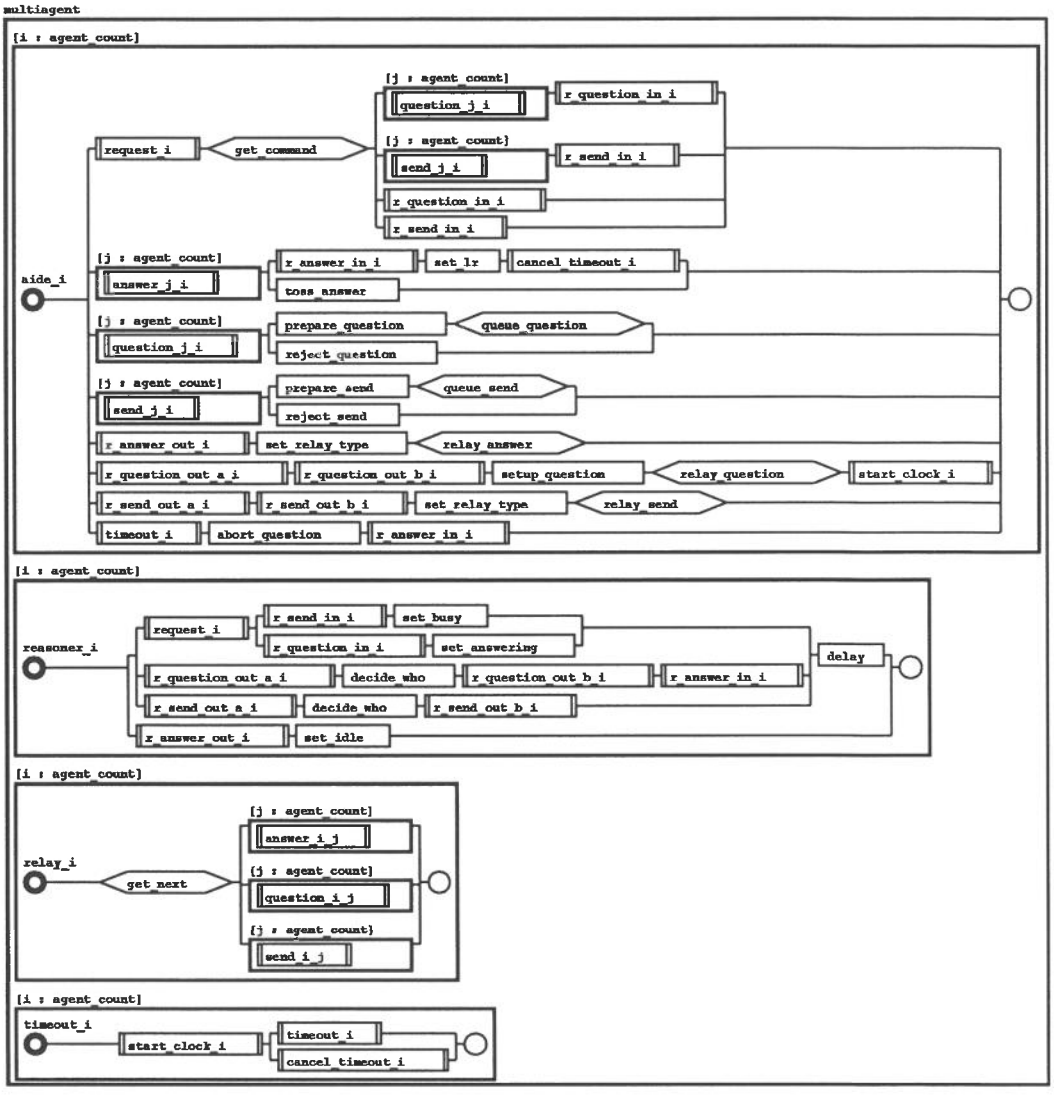

Figure 9: A model of an agent in a multiagent system with error handling

Figure 10 shows the Verdi models of an agent and its environment. Also shown in the figure is a model of a relay process. The rectangular and the "diamond" shaped boxes correspond to various actions and interactions that could be performed by the aide and between instances of the aide respectively. The aide and the relay can then be replicated to create any number of instances of aides and relays. Figure 11 shows a snapshot of the interactions between instances of the aide.

8 Conclusions

When Smith and Davis described the Contract Net protocol [Smith 1980, Davis and Smith 1983] for cooperative problem solving among agents, they listed as a *prerequisite* a means for communicating among the agents. We have specified and implemented an architecture for this prerequisite.

The interagent communication architecture in no way precludes *other* means by which computational agents can interact, communicate, and be interconnected. For example, one agent may be able to view a second agent with a camera, and use the resulting images to coordinate its own actions with those of the second agent. The protocol neither supports nor precludes this kind of interaction. Other interagent communication protocols that have been developed, such as the Contract Net, are defined at a higher level than the method described here, but can be readily implemented on top of this standard.

RAD is a first step toward cooperative distributed problem solving among multiple agents. It provides the low-level communication and reasoning primitives necessary for beneficial agent interactions, but it does not guarantee successful and efficient cooperation. The next steps will require increased intelligence and capabilities for each agent, resulting in more sophisticated agent interactions occurring at a higher level. We are providing these capabilities through our research.

References

[Allen 1987] James Allen, "Belief Models and Speech Acts," *Natural Language Understanding*, The Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1987, Ch. 15, pp. 432–465.

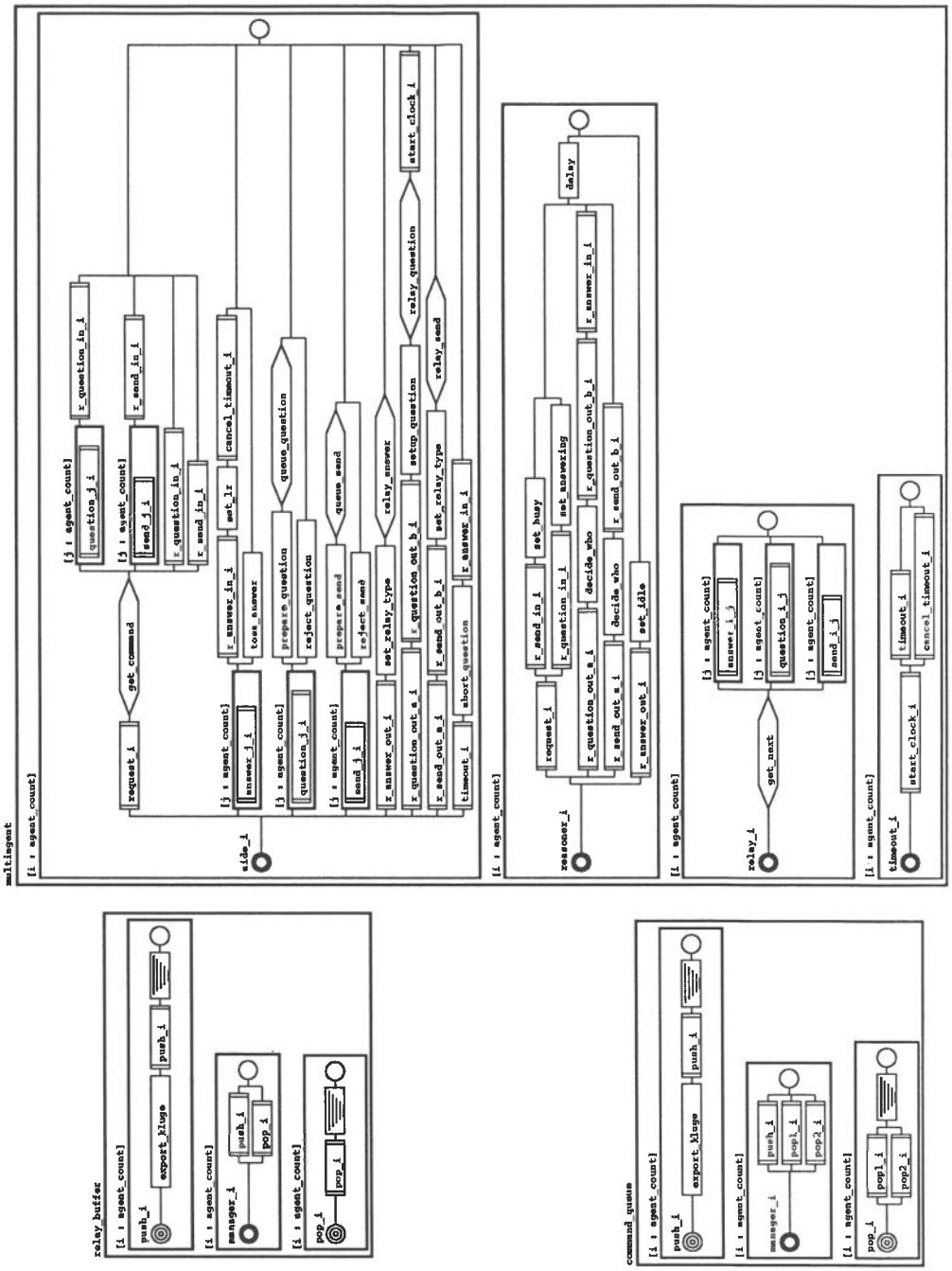

Figure 10: Static models of an agent and its environment

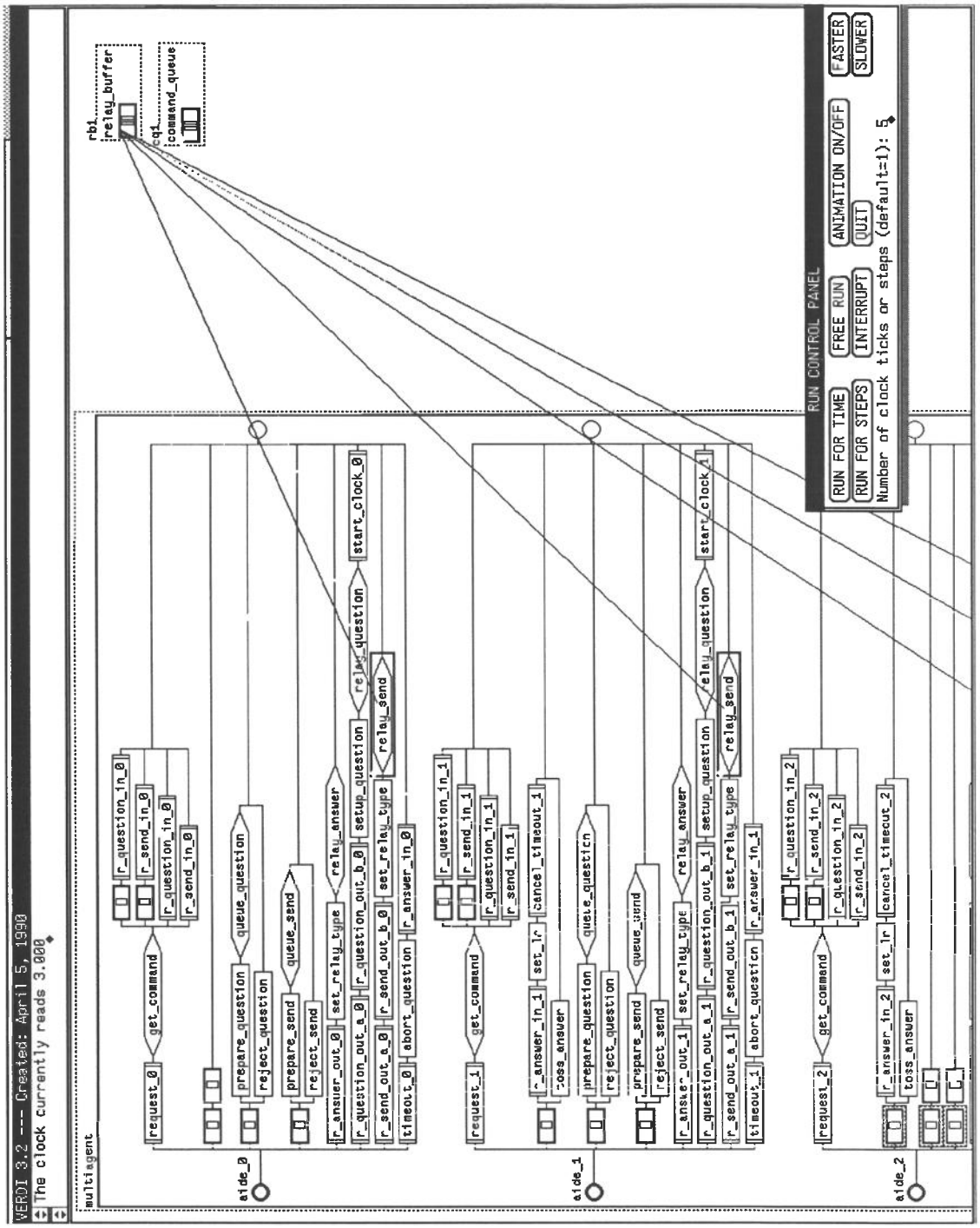

Figure 11: Simulation of interactions among multiple aides

[Arni *et al.* 1990] Natraj Arni, et al., "Overview of RAD: A Hybrid and Distributed Reasoning Tool," MCC Technical Report No. ACT-RA-098-90, Microelectronics and Computer Technology Corporation, Austin, TX, March 1990.

[Bell and Zhang 1990] David A. Bell and Chengqi Zhang, "Description and Treatment of Deadlocks in the Hecodes Distributed Expert System," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-20, no. 3, May/June 1990, pp. 654–664.

[Bond and Gasser 1988] Alan H. Bond and Les Gasser, *Readings in Distributed Artificial Intelligence*, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1988.

[Davis and Smith 1983] Randall Davis and Reid G. Smith, "Negotiation as a Metaphor for Distributed Problem solving," *Artificial Intelligence*, vol. 20, no. 1, January 1983, pp. 63–109.

[Evangelist *et al.* 1988] Michael Evangelist, Vincent Shen, Ira Forman, and Mike Graf, "Using Raddle to Design Distributed Systems," *Proceedings of the 10th International Conference on Software Engineering*, April 1988, pp. 102–111.

[Gasser *et al.* 1987] Les Gasser, Carl Braganza, and Nava Herman, "Implementing Distributed Artificial Intelligence Systems Using MACE," *Proceedings of the Third IEEE Conference on Artificial Intelligence Applications*, 1987, pp. 315–320.

[Gasser and Huhns 1989] Les Gasser and Michael N. Huhns, eds., *Distributed Artificial Intelligence, Volume II*, Pitman Publishing, London, 1989.

[Genesereth *et al.* 1990] Michael R. Genesereth, Thomas Gruber, Ramanathan V. Guha, Reed Letsinger, and Narinder P. Singh, "Knowledge Interchange Format," Logic Group Report No. Logic-89-13, Computer Science Department, Stanford University, Stanford, CA, 94305, January 1990.

[Howard and Rehak 1989] H. Craig Howard and Daniel R. Rehak, "KADBASE: Interfacing Expert Systems with Databases," *IEEE Expert*, vol. 4, no. 3, Fall 1989, pp. 65–76.

[Hsu *et al.* 1987] Ching-Chi Hsu, Shao-Ming Wu, and Jan-Jan Wu, "A Distributed Approach for Inferring Production Systems," *Proceedings IJCAI-87*, Milan, Italy, August 1987, pp. 62–67.

[Huhns 1987] Michael N. Huhns, ed., *Distributed Artificial Intelligence*, Pitman Publishing, London, 1987.

[Melliar-Smith *et al.* 1990] P. M. Melliar- Smith, Louise E. Moser, and Vivek Agrawala, "Broadcast Protocols for Distributed Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 1, January 1990, pp. 17–25.

[Smith 1980] Reid G. Smith, "The Contract Net Protocol: High Level Communication and Control in a Distributed Problem Solver," *IEEE Transactions on Computers*, vol. C-29, no. 12, December 1980, pp. 1104–1113.

[Smith and Davis 1981] Reid G. Smith and Randall Davis, "Frameworks for Cooperation in Distributed Problem Solving," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-11, no. 1, January 1981, pp. 61–70.

[Verdi 1990] Noreen Garrison, "VERDI3 User's Guide," MCC Technical Report No. STP-352-89(Q), Microelectronics and Computer Technology Corporation, Austin, TX, January 1990.