MODEL INTEGRATED PROGRAM SYNTHESIS OF AGENT INTERACTION PROTOCOLS

By

Jonathan Mark Sprinkle

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Electrical Engineering

August, 2000

Nashville, Tennessee

Approved: Date:

_____ _____

_____ _____

To my secret agent,

Mary Margaret.

Namaste

ACKNOWLEDGEMENTS

This research was performed under the sponsorship of the Defense Advanced Research Projects Agency, Information Technology Office, Autonomous Negotiating Teams project, under contract number #F30602-99-2-0505.

I give many thanks to my advisor Dr. Gabor Karsai for keeping me close to the track throughout the development of my research, and especially this thesis. Also, thanks to Dr. Ted Bapty for providing necessary input towards crafting this paper in a way such that those who are unfamiliar with agents can better understand it.

To all the crew here at ISIS: thank you. To Dr. Greg Nordstrom, for his role in my personal steering committee not to mention his benevolent demeanor when passing modeling knowledge along. To Chris van Buskirk for giving much knowledge and understanding of the way agent systems ought to work.

To those who gave me the stamina and perseverance to continue along the way, thanks. To Dr. Roger Haggard, who taught me the meaning of what it was to really do a project, and also taught me that it was indeed possible to be productive for more than 24 hours in one day. To Dr. Carl Ventrice, for being the model professor in action and attitude. To all of my other professors at Tennessee Technological University (Dr. Rajan, Dr. Carnal, et. al.), for inspiring me to pursue a career in academia.

Most importantly, thanks to my parents, Kenneth and Teresa, for never allowing me to think that something cannot be done, and special thanks to Mary Margaret, for whose love and support I am most indebted.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

LIST OF TABLES

LIST OF ABBREVIATIONS

ACL – Agent communication language

API – Application programming interface

CN – Contract net

DSME – Domain-specific modeling environment

FIPA – Foundation for Intelligent Physical Agents

GME – Graphical model editor

IP – Interaction protocol

KQML – Knowledge Query Manipulation Language

MAS – Multi-Agent System

MCL – MultiGraph constraint language

MIC – Model integrated computing

MIPS – Model integrated program synthesis

OCL – Object constraint language

UML – Unified Modeling Language

CHAPTER I

INTRODUCTION

A Multi-Agent System (MAS) is a cooperation focused implementation of multiple programs called *agents* that coordinate with each other to attempt to converge on the solution to one or more tasks. Agent *negotiation* is the convergence upon this solution through compromise and communication. It is the communication portion of negotiation that is the focus of this thesis.

Currently, the implementation of agent negotiation is highly dependent on the programming language in which the agent was developed. Oftentimes, the programmer who implements the negotiation protocol of the agent behavior will solve the general problem of an interaction protocol using constructs native to the agent implementation language. To do this, the programmer is forced to implement a high-level concept using low-level implementation language. He must think about structuring strings, looping structures and temporary variables, etc.

A better solution to the development of agent interaction is to graphically model the negotiation interaction on a high level, and produce from that model the implementation in the agent's native language. Such an approach would remove the burden of language implementation from the agent developer, and also provide a visual guide for other parties interested in the function and organization of the interaction.

Introduction to Agents

Before defining any behavior or organizing any thoughts on interaction protocols of agents, what an agent actually *is* must be established. Unfortunately, it is almost impossible to find one particular definition of the term "Agent." A quick reference to several documents could yield as many definitions as there are documents, not to mention documents that examine the fact that there is no hard and fast definition [1][2][3]. Fortunately, this paper is concerned with the highest levels of agent interaction, so the set of attributes of an agent for this topic is fairly small.

Agents in general,

- can exhibit active as well as reactive behavior, distinguishing them from purely reactive programs, and
- are independently developed.

Agents are non-trivial software components, but despite their complexity they should still be able to communicate with one another in order to solve a problem.

Conceptual Layers of an Agent

In the substance of an agent, several conceptual layers of information exist. The agent "knows" about things (its domain, consisting of objects defined in terms of its ontology), can perform tasks, can communicate, etc. [4]. The definition of tasks and ontologies (or knowledge bases) is typically static to the agent, although some agents with the capability to "learn" are utilized in artificial intelligence applications. For this paper, the most important layer of agent existence is this outer layer. The internal layers of an agent, although extremely important to the overall behavior of the agent, are considered implementation details of the agent environment for the purpose of this thesis.

More importantly, the world outside an agent communicates with the agent through the communication component of this outer layer. It is the communication with the outside world that allows for the dynamic execution of tasks and the acquisition of knowledge. The agent is responsible for interpreting messages it receives, and reacting to them with its statically defined behavior, or with messages to another agent, or both.

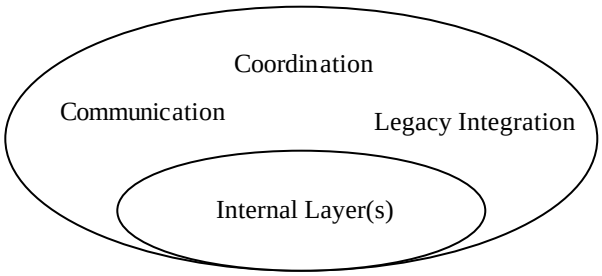


Figure 1. Conceptual drawing of an agent [4]. The internal layers of the agent are accessed by the outer layer through API calls.

Figure 1 shows a conceptual view of an agent at the highest level of abstraction. The communication portion of this outer layer is the most important with regard to negotiation. Coordination and legacy integration are important for the internal behavior of the agent, and thus are not an integral part in defining the structure of an interaction protocol. This view of an agent is consistent with that set forth by the Foundation for Intelligent Physical Agents (FIPA) [5].

Interaction Protocol

The Interaction Protocol (IP) is the sequence of agent messages that allows two (or more) communicating agents to conduct a negotiation. Although there exist standards for the format of messages in agent communication [5], much of the implementation in

general is nonstandard. Agent communication is implemented in many different languages, and the internal workings of an agent are not standardized in any way. Even with the acceptance of the FIPA notion that agents must have communication, coordination, and legacy integration aspects, no standard exists for the implementation of these notions.

Agent systems, although they are different in implementation, are conceptually the same. That is to say, the agent systems all have methods to send and receive messages, link to external databases, gain knowledge of the MAS of which they are a part, etc. The solution point is this: that by conceptually modeling the high-level interaction of an agent, the output can be implementation code. The Model-Integrated Computing (MIC) and Model-Integrated Program Synthesis (MIPS) tools have been proven to provide such solutions [7][8][9][10][11].

The communication portion of an agent is the portion most important to the control flow of an interaction protocol. Therefore, the MIC and MIPS approach to modeling interaction protocols requires a thorough examination and exploration of the concepts of agent communication.

The Relation of Negotiation and Interaction Protocol

Interaction protocols exist, and are used in agent behavior, without the concept of negotiation. Two agents may interact through sending certain types of messages, upon receipt of those messages decide to take a certain logic path, and never converse again. This is an extremely simple interaction protocol. The crux of interaction is made up of the sending and receiving of messages.

An interaction protocol is a necessary but not sufficient component of negotiation. The distinguishing factor between negotiation and interaction is that in negotiation,

agents decide whether to relax constraints and communicate further based on what messages have been received. The art and science of the negotiation method (often called a strategy) is extremely complicated, and involves the internal state of the agent and what knowledge the agent has of its environment.

The negotiation itself is not the focus of this thesis; rather, it is the interaction protocol. However, the place of negotiation is reserved in the modeling environment, because interaction protocols are so useful in defining negotiation protocols; it is merely assumed that knowledge of the negotiation strategy exists external to the model.

Converging on a Solution

In every problem domain, there exists a constant part and a variable part. The constant parts of the problem domain are the concepts native to that domain. The variable parts of the domain are the interactions of the constant parts. In a nutshell, the idea of MIPS is to define concepts of the domain (described in a *paradigm*), and then allow the user to provide the variable part of the problem by modeling it using these concepts. After the problem is modeled, then a custom program interprets the model, and produces the compilable code that implements the solution to the problem.

The objective is to model the domain (in this case, agent interaction) well enough such that as many target environments (or agent implementations) as possible can use the same domain concepts, and to then create a custom program for each target environment. In theory, one would be able to generate solutions for any agent environment with the same paradigm.

Scope

Agents have communication, coordination, and legacy integration aspects. Agents may negotiate with each other through communication, and this communication is generally structured in the form of an interaction protocol. Figure 2 visually describes the relationship of agents and interaction protocols. The interaction protocol is composed of communication concepts that are found in each and every agent implementation, but currently, there exists no high-level process to translate those communication concepts into the form of the interaction protocol implementation code.

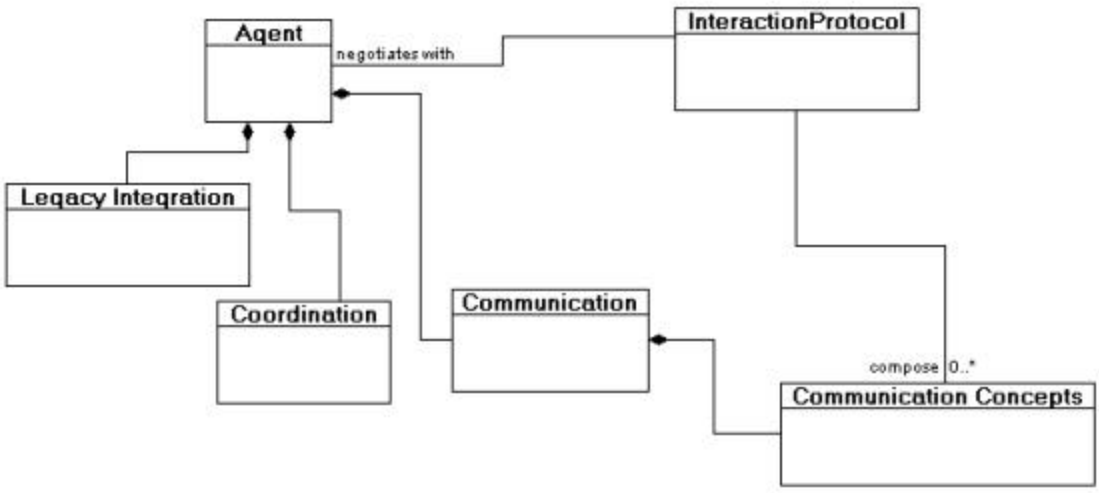


Figure 2. The relationship of the Agent domain to the IP

Thesis Statement

Model-Integrated Program Synthesis may be employed to produce compilable code that implements an agent interaction protocol.

CHAPTER II

BACKGROUNDS

Agent interaction is basically a specialized type of agent behavior. The modeling solution, therefore, should incorporate the concepts of agent behavior that directly relate to IP's. After investigating agent behavior, this chapter then explores the current methods to produce IP's, and then looks at MIPS and what new things it brings to the solution.

Agent Behavior

Negotiation between agents is captured in the behavior of the agent with regard to stimuli. In some respects, therefore, the concepts of agent behavior are used as a subset of the concepts of agent negotiation; thus behavioral concepts must be identified in order to be able to model them. The behavior of an agent may be viewed conceptually in two tightly coupled categories: what it does internally, and how it responds to external stimuli.

Intra-Agent Behavior

Graphically modeling the explicit behavior inside the agent is outside the scope of this paper. First of all, each target application's definition of an agent has different internal capabilities, so by that argument it would not be possible to produce implementations for all target applications because those individual applications would have to be included in the modeling environment.

However, the need to represent internal behaviors during negotiation is recognized and accounted for. The internal behavior of an agent is lumped into a single concept, and treated as a black box that may produce a finite amount of outputs. These outputs may then be used to further guide the negotiation, just as if they were notification from an external source. This thesis recognizes that internal behavior happens, but it is not concerned with how it happens.

Inter-Agent Behavior

Agents communicate with each other through messages, and the sending and receiving of these messages are the two main concepts of inter-agent behavior. Two generally accepted standards for messaging are Knowledge Query Manipulation Language (KQML) [12] and the FIPA Agent Communication Language (FIPA-ACL) [5]. These standards specify certain performatives and parameters. For this particular modeling environment, the FIPA-ACL is implemented as the standard for describing message sending and receiving, and the performatives and parameters are listed in Table 1 and Table 2.

Table 1. FIPA-ACL performatives

Performatives			
accept-proposal	agree	cancel	cfp
confirm	disconfirm	failure	inform
inform-if	inform-ref	not-understood	propose
query-if	query-ref	refuse	reject-proposal
request	request-when	request-whenever	subscribe

Table 2. FIPA-ACL parameters

Parameters	
sender	conversation-id
reply-with	in-reply-to
content	reply-by
language	ontology
protocol	envelope

Messages are composed of these two basic components. A message is structured with a performative (which is required), and any number of parameters that accompany that performative.

Conceptually, at any one time, a negotiating agent is doing one of three things,

- Making internal decisions
- Sending a message
- Waiting to receive a message

The first of these behaviors is accounted for in the agent's internal behavior, and the last two are the main concepts that exist in the inter-agent communication aspect of agent behavior.

Protocol State

One way to model negotiation protocol (or for that matter, any type of behavior) is with a state machine. Before continuing on this line of discussion, it is important to differentiate between the state of the negotiation protocol, and the state of the agent. The negotiation protocol state is determined by the last message received and its content,

while the agent state is a function of the internal values of the agent's variables. In addition, the negotiation state exists separate from the interacting agents. In fact, should one of the agents shut down, the negotiation would still exist, and have a state.

Roles In Negotiation Protocols

There are two distinct roles in any interaction protocol: that of *initiator*, and that of *responder*. Take for instance, the simple protocol of two persons who meet on the street, say, Jon, and Mary.

```
Jon:  "Hello, how are you?"
Mary: "Fine.  How are you?"
Jon:  "Just fine."
```

Then, Jon and Mary resume their paths, or decide to converse further. Jon and Mary both knew when this portion of the conversation was over, because they had a notion of whether they *initiated* or *responded to* the conversation. Consider the following protocol, without this knowledge.

```
Jon:  "Hello, how are you?"
Mary: "Fine.  How are you?"
Jon:  "Just fine.  How are you?"
Mary: "Fine.  How are you?"
Jon:  "Just fine.  How are you?"
Mary: "Fine.  How are you?"
```

There would be no end to the conversation. This example, albeit quite simple, illustrates the need for negotiating parties to understand their roles in the ongoing protocol.

Literature Review

The state machine model of agent behavior is not new to agent development. Several papers and agent building packages point out this implementation scheme, such as the Bond agent system [13], Zeus [14], and COOL [4].

The Bond Agent System

The Bond agent system defines the entire behavior of an agent (not simply the negotiation portion of it) in a multi-plane state machine. In Bond, the overall state of an agent is defined by a vector of states. Each state in this vector is comprised of several nodes, and the state of each plane in the state machine is defined by which node is currently active. The state of an agent changes when a node transitions into another node.

Bond also has the notion of a strategy component. The strategy examines the agenda of the agent, and from the current state machine information and internal agent state determines what the agent's next course of action will be. The strategy component is the Bond implementation of intra-agent behavior.

Bond's agent runtime environment interprets a well-formed multi-plane state machine defined in the Blueprint language, but currently provides no way to implement such a machine other than coding it directly. That is, although the conceptual model of the agent behavior is a state machine, it must be implemented using procedural code.

A negotiation protocol in Bond is merely one of the planes in the multi-plane state machine. It may be included along with the other behavior which is specified with the Bond agent development software.

The Zeus Agent-Building Toolkit

Zeus, like Bond, incorporates the behavior of the entire agent into a state machine format: specifically, a directed graph. Zeus agents pursue a goal, and to achieve that goal execute behavior defined along a directed graph that best describes what they wish to do (e.g. buy, sell) to achieve that goal.

Zeus agents follow a statically defined graph structure. The graph that an agent traverses is composed of what Zeus calls nodes, arcs, and sub-graphs. Each of these parts of the graph define behavior of an agent, and when the behavior is completed, the graph passes the state information of the agent on to the next node or graph. The state information is examined on exit of each graph part, and the results of that examination determine what portion of the graph will next be executed. In this sense, the agent traverses the graph by visiting nodes until reaching an endpoint. These graphs are predefined for all agents, and therefore need not be implemented to give behavior to an agent.

The agent uses the state information contained within the graph to determine whether it should converse with another agent to achieve the goal. Should the agent require assistance from another agent, then an interaction protocol is used for the conversation.

In Zeus, a negotiation protocol is one of the graphs executed in the statically defined graph structure. At agent construction time, the agent developer specifies the negotiation protocol, and then compiles the agent. Therefore, to produce a negotiation protocol in Zeus, a graph must be produced.

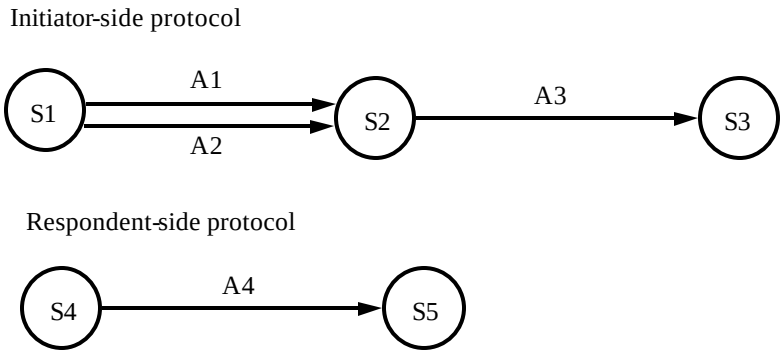


Figure 3. Sample Zeus implementation of the contract net protocol

Table 3. Description of Initiator in Figure 3

Node/Arc	Description/Transition Condition
S1	Identify agents that can perform goal
A1	Select subset of agents that can perform goal and who are co-workers (check $\neq \varnothing$)
A2	Select subset of agents that can perform goal and who are peers (check $\neq \varnothing$)
S2	Send request for proposals to selected agent and await responses
A3	Check that an accept response has been received
S3	Done

Table 4. Description of the Responder in Figure 3

Node/Arc	Description/Transition Condition
S4	Evaluate cost Send accept message Await response
A4	Contract award message received
S5	Done

Figure 3 and Tables 3 and 4 show an example conceptual implementation in the Zeus framework. Although when a Zeus agent executes it traverses a well defined graph (similar to a state machine), the negotiation protocol itself is but one node in the graph. This means that, like Bond, the Zeus negotiation protocol is conceptually a state machine, but it is implemented through procedural code.

COOL

The Coordination Language (COOL) implements negotiation through what it terms "conversations." COOL also views negotiation as a finite state machine, and uses the concept for visualization and conceptual organization. Figure 4 displays the FSM representation of a COOL conversation.

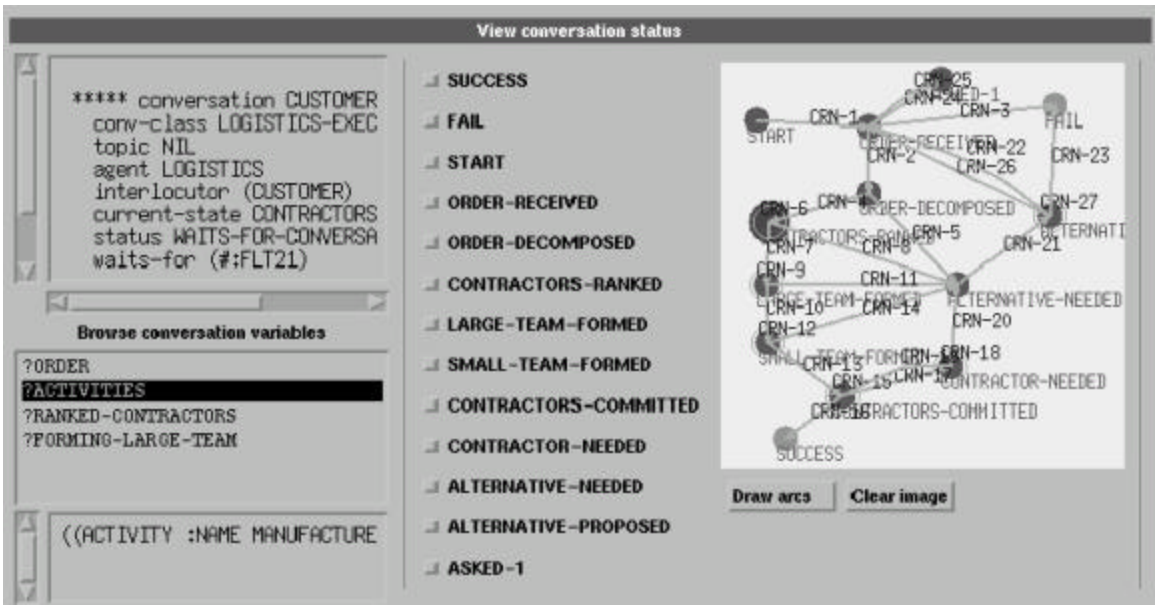


Figure 4. Visualization of a COOL conversation.

COOL accepts a structured definition of the state machine as behavior. An advancement (in comparison to the previous implementations) is that the implementation code of a COOL protocol is not procedural, but is instead reminiscent of a state machine. That is, a state machine syntax is used, rather than a procedural layout of case statements. Figure 5 exemplifies the COOL syntax for creating the behavior shown in Figure 6.

```
(def-conversation-class 'customer-conversation
    :name 'customer-conversation
    :content-language 'list
    :speech-act-language 'kqml
    :initial-state 'start
    :final-states '(rejected failed satisfied)
    :control 'interactive-choice-control-ka
    :rules '(  (start cc-1)
               (proposed cc-13, cc-2)
               (working cc-5 cc-4 cc-3)
               (counterp cc-9 cc-8 cc-7 cc-6)
               (ask cc-10)
               (accepted cc-12 cc-11)))
```

Figure 5. COOL structured language to describe the agent behavior of Figure 6

Like Zeus, COOL provides a state machine visualization of the agent behavior at runtime. However, like each previous implementation it provides no development tool that allows the agent designer to conceptually model the conversation and produce from that model the implementation code.

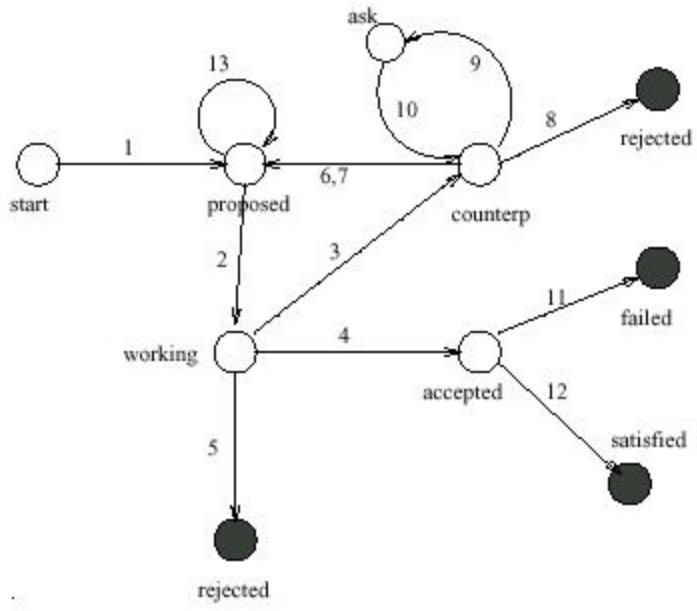


Figure 6. State machine graphical representation of a COOL behavior

Other Approaches

It is not feasible to examine in this thesis all of the available agent packages and their shortcomings. However, it is worth noting that there exist more packages than Zeus, Bond, and COOL, but that the three of these are a representative sample of the agent packages that consider the interaction protocol as an integral part of agent behavior. Other packages are very similar to hese three in that they provide functionality to input an interaction protocol into the behavior of the agent.

However, there are packages such as Mad-Kit [6], which do not even recognize the problem of the interaction protocol. In packages such as this, all of the behavior is implemented with low level computer languages such as Java or C/C++, and any interaction protocol must be added as a layer by the developer. It is the goal of this thesis to provide a solution both for agent packages of the kind that acknowledge the presence of an interaction protocol, as well as those that do not.

Literature Conclusions

There are several implementations of agent frameworks that use state machines to describe behavior. However, the usage of FSM in the development of behavior (specifically, negotiation) is lacking. Some of the approaches accept a structured definition of the behavior state machine, but none of them provide a tool with which to take advantage of the conceptual nature of a state machine (i.e. graphically associate states and transitions). The best any of these approaches do is to allow the agent developer to visualize the current state of the conceptual state machine using custom designed tools for that approach. To implement any of the behavior, the modeler is forced to think on a low-level to produce the high-level effect.

Furthermore, even if one of the approaches *had* provided such a tool, any implementation code the solution would have generated would be limited to use within that agent environment.

MIC and MIPS

The solution to the problem of modeling agent interaction protocols must

- visually present the solution during development,
- present domain concepts that are specific to interaction protocols, not the agent environments, and
- produce implementation code to allow the agent environments to utilize the graphically specified behavior.

MIC and MIPS enable a tool to satisfy all three criteria. The nature of a graphical modeling language satisfies the first criterion. The next two are satisfied through the modeling paradigm, and the interpreter.

The Paradigm

The Graphical Modeling Environment (GME), developed at Vanderbilt University, is a tool that allows for the development of Domain Specific Modeling Environments (DSME's), and domain specific models. The *paradigm* defines the entities and relationships of a particular domain, and maps those allowed entities and relationships onto generic modeling concepts within the GME [15].

Together with the GME, the paradigm is the specification for a graphical modeling language. This language has a syntax, with "performatives" such as models, atomic

parts, connections, references, and attributes for all of these performatives. For an in-depth study of modeling concepts, please refer to [15], and to Appendix A.

The Interpreter

The interpreter's role in the solution is the production of agent implementation code. The paradigm defines the syntax of the modeling language, and the static semantics (via constraints), but it is the interpreter that gives semantics to the organization of the actual model instances. Just as textual programming languages have a compiler, the graphical language has an interpreter.

When defining the interpreter, behavior is associated with each concept and its relationships with other concepts. At interpretation time, the graphical model is translated into an implementation within the domain (in this case, agent negotiation).

The interpreter is the part of the modeling process that is not domain independent. In fact, to produce implementations for *n* agent environments would require *n* interpretations of the models. However, if the modeling environment is used often, then the time necessary to write the interpreter is a small price to pay compared to the time that would have been spent writing the implementation code.

CHAPTER III

MAPPING CONCEPTS INTO THE MODELING ENVIRONMENT

To create the paradigm for the agent negotiation domain, the domain concepts must be analyzed and mapped onto the most appropriate modeling concept (e.g. model, atom). This chapter assigns domain concepts to modeling concepts, and gives justification for the assignment. The paradigm is the MIC portion of the solution, and once the domain concepts are established in it, then it is a domain-specific modeling environment (DSME).

There are two levels of mapping that must take place in order to have a complete domain specific modeling environment for agent negotiation. The first level revolves around precisely defining the concepts of agent negotiation. The second level deals with designing the modeling environment is such a way that at modeling time, the environment provides a visual representation of a state machine.

There is also an overall mapping that can give visualization to the interaction between roles in a protocol, and provide for a behavior that links protocols to one another.

Agent Negotiation Concepts

As described in Chapter II, there are three basic concepts in agent behavior: intra-agent behavior, sending a message, and waiting on a message. The names by which the paradigm refers to the concepts are in bold type, and are capitalized throughout the remainder of the document when they refer to types within the paradigm, and not just types within the agent domain. Figure 7 shows an abbreviated class diagram of the basic con-

cepts of the paradigm. For the complete diagram that was used to synthesize the paradigm refer to Appendix C.

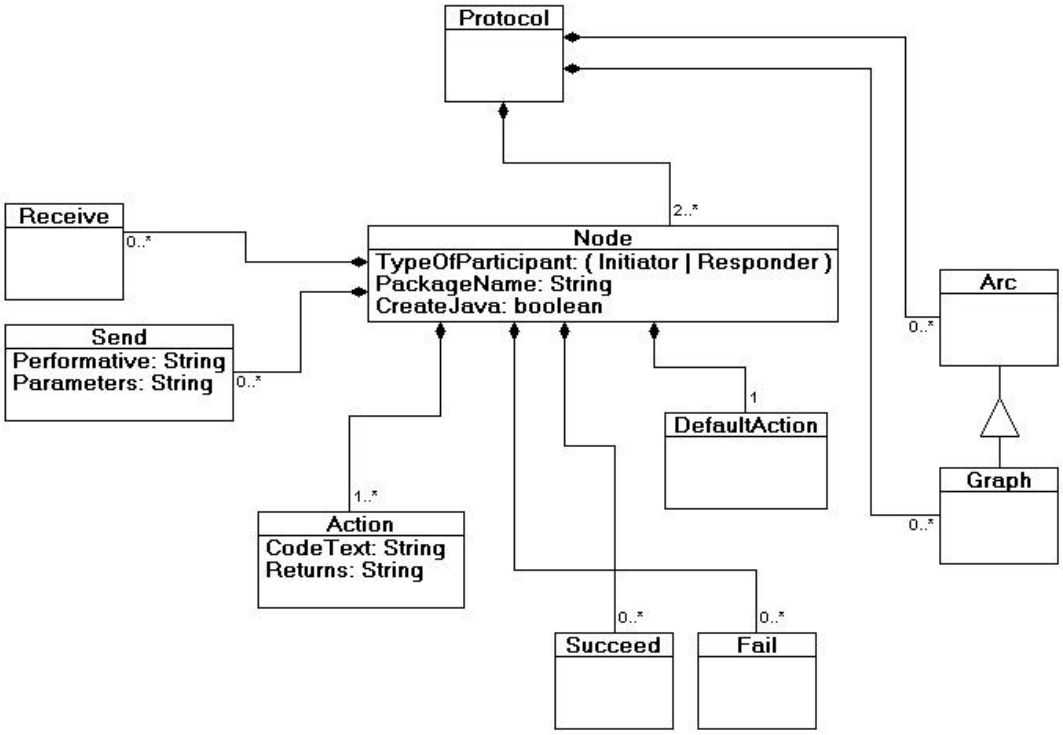


Figure 7. Simplified UML class diagram of the paradigm

Intra-Agent Behavior

Intra-agent behavior is treated as a pre-existing black box that can produce a finite set of results. Since there is no containment or sub-behavior associated with intra-agent behavior, it is modeled as an atomic part, and given the name **Action**.

There are two attributes of an Action atom. The first is the code that is associated with the execution of this Action at runtime. This code is highly implementation specific, and is the most time-consuming part of the protocol to compose. The second attribute of an Action atom is the finite set of return values that the code has. These return values will be used to determine the next stage of the interaction protocol.

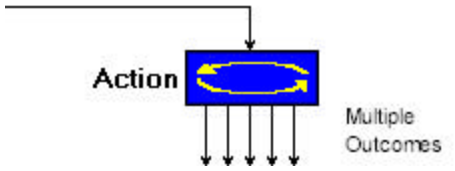


Figure 8. The Action atom may have multiple result paths

The return values are somewhat more implementation independent than the code attribute of an Action. They represent on a more conceptual level the logical function of the Action, and the possible answers at which the Action may arrive based on the internal state of the agent and the state of the negotiation. For example, regardless of how an Action is implemented, if it is to perform the task of evaluating a proposal, it will return values of "acceptable" or "unacceptable," and perhaps some variant on these. The visual model of the interaction protocol is unaware of the logic of the Action, but is aware the Action will produce a finite set of answers, as shown in Figure 8. The negotiation protocol cares only what the Actions tell it.

Sending a Message

The concept of sending a message is also represented as an atomic part, and is referred to in the paradigm as **Send**. Each time a message is sent, it must have as attributes the performative type, as well as any parameters that accompany that performative. The listing of performatives and parameters was discussed in Chapter II.

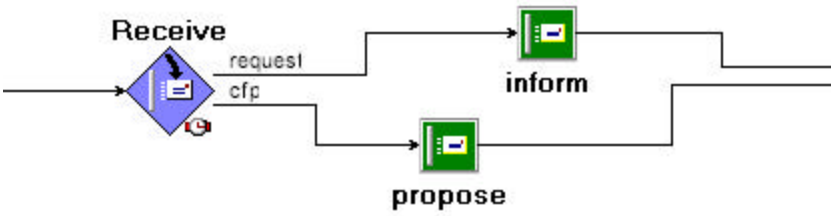


Figure 9. Sending different messages based on message receipt type

Receiving a Message

The receipt of a message is also considered as an atomic part in the domain, and is referred to as a **Receive**. A Receive atom signifies that the interaction protocol is awaiting a message. However, as more than one message may be plausible at any one time, the logic to take a certain path based on a certain message lies not with the Receive atom, but with the transitions out of it. That is, a Receive can have multiple behaviors, depending on its input, so the attributes of the Receive are located in the connections emanating from it. Figure 9 displays the behavior design of sending different performatives based on the last message type received.

Returning the Result

After the negotiation is completed, it must have a way to notify the caller of the degree of success of the negotiation. Therefore, two more atoms exist, called **Succeed**, and **Fail**. These two atoms are aptly named, and in general represent the end of the negotiation specification. Figure 10 provides an example usage of the Succeed and Fail atoms.

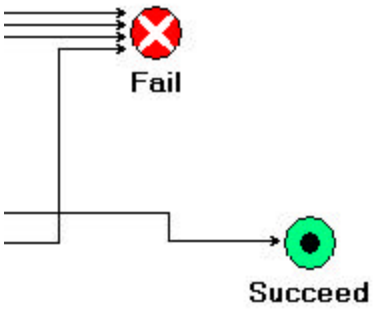


Figure 10. The Succeed and Fail atoms signify the end of the behavior definition

Agent Negotiation Node

All of the concepts that make up a negotiation are contained in a model. The negotiation node is the container of all the behavioral atoms of the negotiation. Therefore, the node is represented as a model, called a **Node**, with the Action, Send,

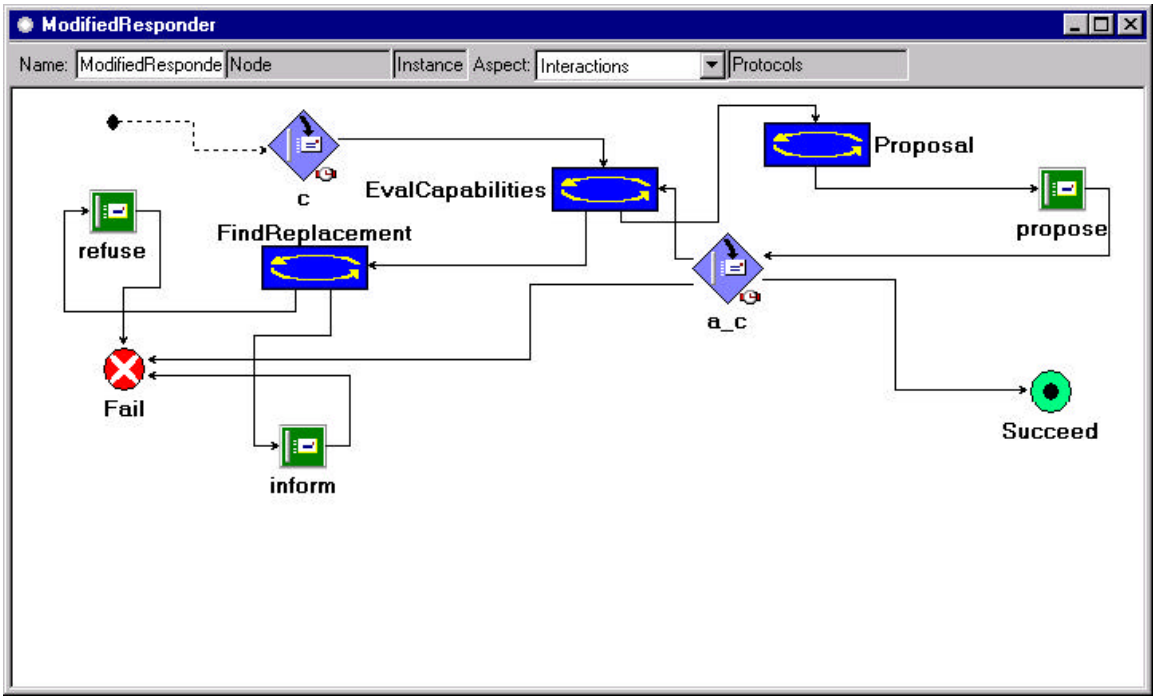


Figure 11. An example Node model, complete with parts.

Receive, Succeed, and Fail atoms as parts. See Figure 11 for an example of the structure of the parts within a Node.

The Node model has several attributes. There are several housekeeping attributes, such as the name of the class which is to be created, and whether or not to create an output file at all for this Node. The reason for this latter attribute, is that there may already exist a Node to which this protocol may want to link; however, there would be no need to generate code for that Node.

As previously mentioned, each agent taking part in a negotiation must have some concept of what role it plays in that negotiation (e.g. initiator or responder). Therefore, a role attribute with the two possible values of "Initiator" and "Responder" is part of the Node model.

The last attribute is a listing of class-wide variables for the Node. This exists because several of the Actions may wish to share state information. Therefore, global variables are permitted, and specified in the Node. These variables may be as specific as the Action code, and must therefore be expressed textually.

State Machine Concepts

A state machine framework was designed that used as states and transitions the concepts from intra-agent and inter-agent behavior. The framework follows the concept of a Mealy state machine [16], with the states identified with waiting for a message, and the behavior of sending a message or performing some action within the state of the agent viewed as a transition. The state machine framework gives graphical presentation to the interaction protocol, which is structured using the domain specific concepts.

Possible States and Transitions

For the purpose of this thesis, the only concept of waiting in agent negotiation is that of the Receive. It is possible to wait on as many types of messages as a send message action could send (in accordance with the FIPA-ACL). The transitions extending from the wait message therefore have an attribute that specifies the performatives upon which this Receive atom is waiting. At model building time, for each performative type for which the Receive is waiting there will be exactly one transition extending from the Receive. The transitions of the state machine are mapped into the modeling environment as connections. Figure 9 displays two transitions from a Receive atom, one named "request", and the other "cfp".

Possible Negotiation Outputs

Along the transitions from Receive to Receive (state to state) are the outputs. The two types of outputs are the Send and the Action atomic parts. Of these two atoms, only the Action has attributes for its connections. There is a specific behavior that is associated with the Send atom, and only one transition is possible once the Send atom is processed in the state machine. However, the Action atom may direct the state machine in a number of finite directions (specified by its return variable attribute, as described in Chapter II and displayed in Figure 8). Therefore, the connections coming from an Action atom contain as an attribute the name of the return value which directs the state machine along this path.

The Default Action

As with any state machine, the entry point must be specified. The atom to specify this is the **Default Action** atom, shown in Figure 12. There may be only one Default Action per Node, and it may connect to any atom in the Node. That atom to which the Default Action points is executed upon entry.

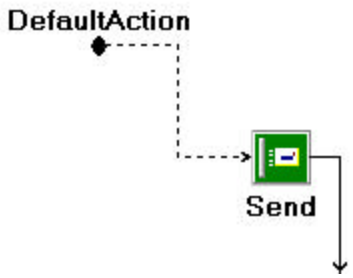


Figure 12. The Default Action

The Overall Protocol

The Protocol contains one or more Nodes. As the Protocol is a container, it is therefore represented in the paradigm as a model. Each of the nodes represents a role in a negotiation. There is no restriction on how many Nodes that a Particular node may interact with. In the Protocol model there are two aspects: the State Interconnection aspect, and the Graph Layout aspect. Also, several atoms exist in the Protocol model that play the role of structuring the Protocol layout, namely the **Arc** and the **Graph**.

The State Interconnection Aspect

In theory each Receive transition in one role (either initiator or responder) should have a corresponding Send in the other role. Otherwise, the protocol could end up in a

state from which it would have no exit. In this aspect, therefore, it is possible to match states and transitions in one Node to those within another Node as shown in Figure 13.

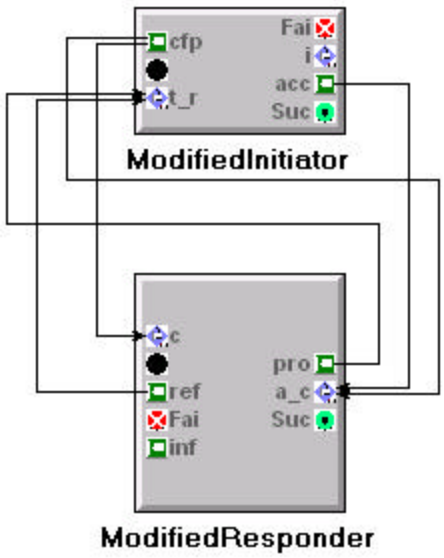


Figure 13. The State Interconnection aspect of a Protocol containing two Nodes

It is the responsibility of the modeler to take advantage of this feature of the modeling environment; it is not strictly enforced or constrained by the paradigm. The reason for this is that it is not an error to have Receives that do not match to a Send, because of implementation details, or special requirements of the interaction protocol. Hence, this ability to match aids in omission errors, but does not restrict the modeler's ability to custom craft the agent negotiation behavior.

Sends and Receives are visualized in this aspect as ports in the Node model, to allow for interconnection. Naturally, since the interconnection of Sends and Receives in this aspect is not enforced, there is no semantic meaning for the connections (although they may be used later for model verification).

The Graph Layout Aspect

This aspect has an overall semantics of the behavior of an agent with respect to its interaction protocol. The aspect is provided to give the modeler the opportunity for code reuse of predefined or already modeled negotiation protocols. For example, if the failure of one protocol necessitated the execution of another, it could be specified in this aspect. Similar to the Node, the organization of this aspect of the Protocol model is similar to that of a state machine. The aspect visualizes Arcs and Graphs. Refer to Figure 14 for an example of the Graph Layout aspect.

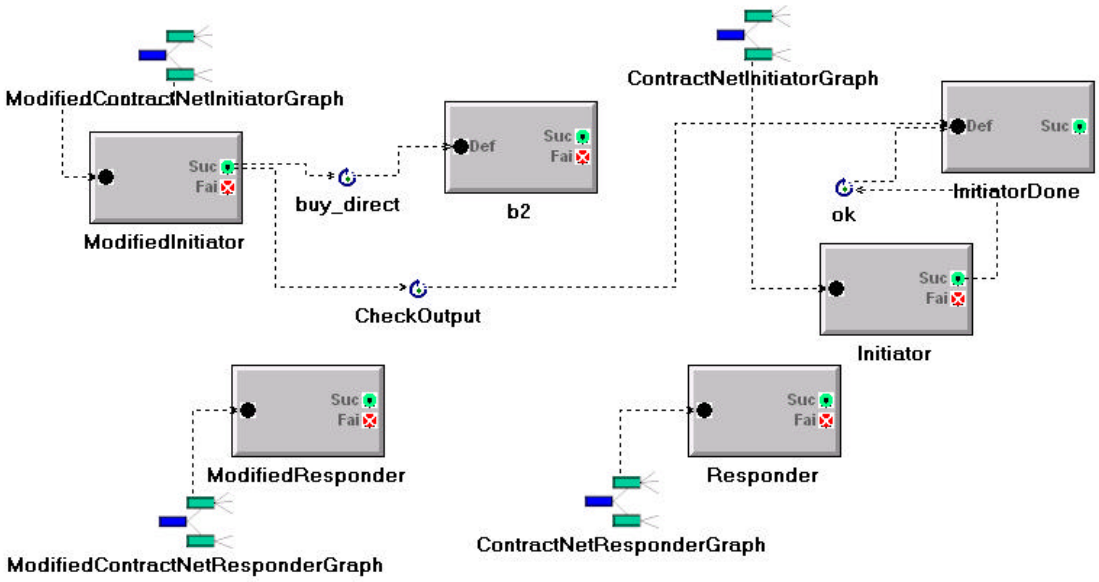


Figure 14. The Graph Layout aspect of a Protocol model, containing several Nodes, Arcs, and Graphs

The Arc

The Arc atom allows the connection between the return values of one Node, and the Default Action of another node. Because the result of the negotiation is the output of a Node, the Arc atoms must be sophisticated enough to examine the state information passed from the negotiation to determine whether the negotiation was a success or a fail-

ure. After all, success to one protocol may be failure to another. Therefore, as the implementation of an Arc must be fairly complex, it is modeled similar to Actions, in that it provides a text attribute that allows for the input of Code. The Arc must return a boolean value of either true or false. Examples of Arcs in Figure 14 are "buy_direct", "CheckOutput", and "ok".

The Graph

The Graph is an atom that groups together one or more Protocols and Arcs and combines them into one conceptual protocol. The Graph is a conditional controller [15], which means that it operates like a set container (see Appendix A for a detailed description of a conditional controller). The purpose of the Graph is to define what Node models will actually be implemented in code as interaction protocols.

The Graph must own not only what Node models are to be included, but also the Arcs that connect them. This is to allow for reuse of Arcs between Graphs. For example, two Protocol models may have more than one Arc that connects them. The different Arcs may have different logic, and one Graph may wish to use one Arc as opposed to another. This degree of control is offered when the Graph atom conditionally controls both the Arcs and the Nodes.

To specify which Node model is the first in the interaction protocol, the Graph connects to the Default Action port of that Node model. An example Graph in Figure 14 is "ContractNetInitiatorGraph".

Constraints

Constraints are enforced by GME at the user's request, and are expressed when the paradigm is defined. One of the constraints of this modeling environment is that exactly one default action is allowed per Node, and it must connect to one and only one atom.

Another constraint is that a send atom must have exactly one connection coming out of it (to ensure that the state machine is deterministic). See Appendix C for a listing of all the constraints.

CHAPTER IV

MAPPING THE DOMAIN CONCEPTS INTO AN IMPLEMENTATION

The domain concepts, now in place in the paradigm, constitute a domain-specific modeling environment. The contents of the domain-specific modeling environment (DSME) all have a semantic role in the representation of the agent interaction and negotiation domain. The implementation of the interaction protocol, which is the output of the MIPS portion of the solution, is assigning a semantics to the models created in the DSME by translating them into code.

The mechanism that performs this translation is called an interpreter. Since the output of the model interpreter is specific to the agent implementation, there must be one interpreter for every agent package. The mapping of concepts to behavior, whether through direct translation or inference of semantics, is different for each interpreter. Therefore, for the sake of brevity, only one interpreter architecture will be described: the one for the Zeus toolkit.

Zeus In More Depth

As mentioned in Chapter II, the behavior of Zeus agents is defined through a directed graph. The runtime environment that traverses this graph and its nodes is known in Zeus as the *coordination engine*. The coordination engine handles examining the definition of the graph, determining what node to execute, and executing that node. The negotiation protocol is one of the nodes in this graph, and it is defined in its own sub-graph

of the main graph. It is necessary, therefore, to understand the components of a Zeus graph, and what it must contain before a negotiation protocol may be produced in Zeus.

Some of the names of the components of the Zeus solution coincide with those of the paradigm, so the Zeus concepts are written in italics.

The Coordination Engine

The Zeus coordination engine is a manager of as many different *Graphs* as the agent is currently executing. Each *Graph* relays state information to the coordination engine, and the engine responds by putting some *Graphs* to sleep, and waking up others.

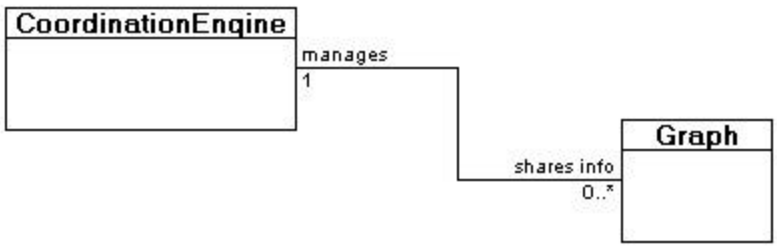


Figure 15. The coordination engine, and its association with *Graphs*

The coordination engine manages the control flow of the behavior through a structure defined in the *Graph* files, as shown in Figure 15. This structure is a state machine listing of *Nodes*, *Arcs*, and other *Graphs* that define further behavior. Each of these types of classes has a special definition within Zeus, but the thing they have in common is that they all implement the function `exec( )`.

The coordination engine calls the `exec( )` function of whatever portion of the directed graph is next. Control is returned from that *Node*, *Arc*, or *Graph* when the `exec( )` function returns a value. Depending on the value returned, the coordination engine di-

ther continues to call methods of this same object, or it moves on to the next portion of the graph. *Arc* and *Graph* implement `exec( )` only, but *Node* also implements the `continue_exec( )` method.

The Node Class

The *Node* is the most powerful member of the coordination graph layout, simply because its `continue_exec( )` method allows it to be executed until some internal state changes. The coordination engine will call the `exec( )` method of the *Node* class when the *Node* is first executed. Thereafter, the coordination will call the `continue_exec( )` method until it returns a specific value (`OK`, or `FAIL`). Before returning one of these two terminal values, the `continue_exec( )` method may return the value `WAIT`, in which case the coordination engine will put this graph to sleep until a new message is received.

The *Node* may access the state of the agent through a special context variable that initializes the *Node* to an executable state. Through this context, the *Node* can access such things as the ontology, resources, community knowledge, as well as the global agent system time and other MAS attributes.

Necessary Interpreter Outputs

Recall that in Zeus, the negotiation protocol is a substitution into the behavior graph of the Zeus agent. Examining current protocol implementations yields the following list of requirements.

- The negotiation is defined in a *Graph* class, which consists of at least one *Node* class, and may be specified as a directed graph of several *Node* classes.

- The *Node* class must extend the *Node* class defined by Zeus, which means it must implement two methods, `exec( )`, and `continue_exec( )`
- The *Node* class gets its idea of the MAS through a context variable which is passed to the *Node* through its constructor
- The *Node* class must present as its output a variable of type `LocalD-Struct`, which is defined by Zeus

These are the major requirements for the output classes. The most involved of these classes is undoubtedly the *Node* class, because it is in the *Node* class that the logic of the interaction is captured.

Low-Level Mapping of Paradigm Concepts to Behavior

Some concepts in an instance of the DSME directly translate to implementation code with certain semantics, regardless of the context in which the concepts are placed. These are the basic concepts, which were described in Chapter III in the paradigm. Table 5 gives a brief description of the mapping that takes place for these models and atoms explains some of the lower level behavior of the interpreter that produces the final implementation.

Table 5. Mapping of concepts to low-level outputs

Paradigm concept	**GME Type**	**Low-level mapping**
Protocol	Model	The *Graph* class in the Zeus environment.
Arc	Atom	A transition from one *Node* to another in the *Graph*
Node	Model	The *Node* class in the Zeus environment
Send	Atom	Writes a string to execute a Zeus API method in the *Node* of which this Send Message is a child. The actual name of the method depends upon the context in which the Send atom is placed. However, the attributes written remain constant, and are obtained from the atom.
Receive	Atom	Returns control of the agent to the coordination engine, who will notify the *Node* when it has a new message. Upon receipt of a message, the Receive atom translates into a pattern matching `if-else-elseif` structure. The clauses for the matching directly correspond to the attributes of the connections coming out of the Receive atom.
Action	Atom	In the protocol implementation, writes itself in a decision format similar to that of the Receive atom. 1. It executes the code associated with it as an attribute. 2. It interprets the return value of the code, and determines (based on the attributes of the connections coming out of it) which logical path to take next. The code of the action is written as a private function of the *Node* class of which it is a child, and it returns an integer value.
Succeed	Atom	Sets values of the output variable, and returns control of execution to the coordination engine.
Fail	Atom	Sets values of the output variable, and returns control of execution to the coordination engine.

High-Level Mapping of Model Instances Through Context

The high-level mapping of the implementation derives semantics from syntax, infers semantics from visualization techniques, and draws upon implementation specific

information of the target environment to determine other high-level outputs. This mapping produces target agent environment specific implementation details such as what values to place in the class constructor, whether low-level mappings like Send and Receive should be arranged in a special way, etc.

If the interaction protocol modeling environment were designed specifically for Zeus, then the domain concepts would be Zeus specific. In this case, there would be a low-level mapping from the concepts to the output. However, the objective is to produce Zeus specific code from a non-Zeus specific environment, and therefore, some of the information that Zeus requires may need to be inferred from the modeling environment. Therefore, it is convenient to examine the requirements of Zeus, and then examine the modeling environment to see how to best extract the required information from it.

The Node Class

There are several high-level concepts within the Zeus *Node* class that require special techniques in implementation. The main caveats deal with the structure of the two methods of the class, `exec( )`, and `continue_exec( )`. When the Zeus runtime engine executes a protocol, it calls the `exec( )` method. After `exec( )` is called, the Zeus runtime environment calls the `continue_exec( )` method until the negotiation protocol returns a certain value.

Mapping Behavior to the Class Methods

The modeling environment has no way to directly map whether the runtime engine should be calling the `exec( )` or `continue_exec( )` method; if it did, then the solution would be more Zeus-dependent than is perhaps necessary. However, in

lution would be more Zeus-dependent than is perhaps necessary. However, in examining how the runtime environment calls the negotiation protocol, a solution may be found.

The negotiation protocol is put to sleep each time it waits for a message, and when an incoming message is received by the Agent, it notifies the protocol by calling the `exec( )` or `continue_exec( )` method. The key for the implementation is that after the first Receive the agent runtime environment will call `continue_exec( )`. When writing the Java output file, then, the interpreter realizes that everything will be written inside the `exec( )` method, until a Receive atom has been written. After that point, the logic is inserted into the scope of the `continue_exec( )` method.

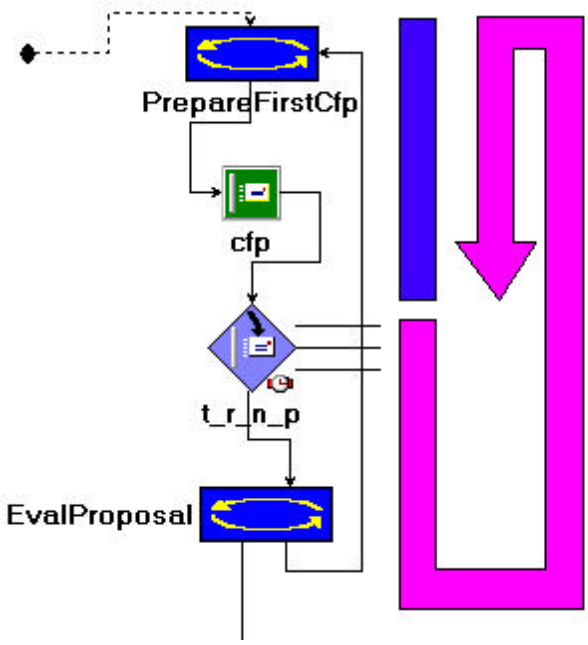


Figure 16. Changing of destination method from `exec( )` to `continue_exec( )`

Figure 16 uses a disjoined arrow to represent the changes of the class methods. As shown by the darker line, the `exec( )` method receives all logic patterns until the first Receive. As the coordination engine will now be calling the `continue_exec( )` method,

it is the appropriate destination for all logic, even when looping through logic that was executed already in the `exec( )` method.

Mapping Dialog Inside the Class Methods

The other major high-level concern within the *Node* class is that of continuing the conversation. In Zeus, it is the `new_dialogue( )` and `continue_dialogue( )` methods that simultaneously send a message and signal the runtime-environment to set this negotiation to sleep. As may be expected, the `new_dialogue( )` and `continue_dialogue( )` methods are similar to the `exec( )` and `continue_exec( )` methods in that the higher levels of Zeus interpret the methods differently. However, conceptually to Zeus it is still the same thing; sending a message and giving up control of the process (going to sleep). As a side note, this is another justification of the concept of modeling protocols on a high level, and allowing the interpreter to perform the implementation.

Similar to the execution methods, therefore, the context of the Send icon with relation to the Default Action and the first Receive is considered when deciding which method to write. Another consideration is the role of the Node in the protocol. Only the initiator role begins a new conversation, because the conversation already exists (i.e. has a conversation identification number within the Zeus environment used for message routing) when the responder role receives the message.

Dialog Existence In a Zeus Negotiation Protocol

In Zeus, the conversation already exists when a responder protocol is created. This implies that the message has already been received by the responder Agent. Naturally, the Agent does not expect another message immediately; it must first provide some logic to the negotiation. However, all agent implementation architectures may not work

like this; in fact, some Agent architectures require that an Agent always be listening for a certain kind of message, while in Zeus, it is assumed for the beginning of a protocol that a certain kind of message must have already been received. Therefore, in the responder Node of a Protocol, the Receive atom has no semantic meaning until at least one message has been sent.

Rather than create an error if a Receive atom is placed before the first send, the Zeus interpreter merely alerts the user that the atom is out of context, and will therefore be ignored in the output code. In this way, the same model may be implemented later using another agent architecture as the target output, and the model will not have to be redrawn, but can use this Receive atom that Zeus deems of no consequence.

The interaction protocol is also awakened when a message has not been received for a certain timeout period, so the timeout connection of the Receive atom is translated as an execution of the timeout logic of the *Node* class.

Managing Multiple States Through a Single Entry Point

The entry point to the *Node* when a message is received is in one and only one place (the `continue_exec( )` method). This can provide a problem if a negotiation wants to have two or even three separate message waiting states. To solve this problem a special mapping takes place. In the `continue_exec( )` method, the very first logical block examines the message that was just received. Therefore, all possible messages this protocol can receive are listed in this logical block. Figure 17 shows two Receive atoms that together expect four possible messages. Therefore, in the logic block at the beginning of the `continue_exec( )` method the message received will be tested against all four possible types to see if there was a match.

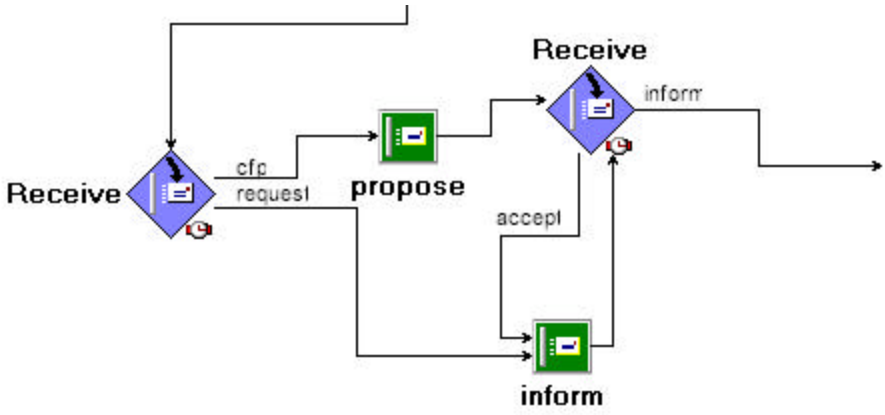


Figure 17. Multiple Receives are mapped to the same logic block

The Graph Class

The Zeus *Graph* class directly translates from the model. The state machine resemblance of the Graph Layout aspect of the Protocol model lends itself well to the construction of the Graph class, which is laid out as a directed graph in the *Graph* Java class.

Overall Interpreter Logic

The basic algorithm of the interpreter is a combination of traversing the possible connection paths from atom to atom in a Node model, and translating the context of each of those atoms into an acceptable Zeus method call with proper semantics. The pseudocode for the interpreter is found in Figure 18.

```
Loop through each Protocol model
   Get all models and atoms in the model
      If Node model
         Write Node class
            Create output file and write header and constructor
            Declare global variables
            Declare return values of Action atoms
            Fill in exec( ) method
               Setup local and timeout variables
               Get the Default Action of this Node
               Write the translation of atoms connected to the Default Action,
               And its further connections until the first Receive is encountered
            Fill in continue_exec( ) method
               Setup local and timeout variables
               Setup received message retrieval
               Get all Receive atoms, and write out their logic matching
                  Write the translation of the atoms connected to the
                  Receive atom, and its further connections until either
                  Another receive is encountered, or a Succeed/Fail is written
            Get all Action atoms
               Write the Code attribute of each Action atom as a private method
      If Graph atom
         Write Graph class
            Create output file and write header and constructor
            Write directed Graph specification
               Get all members of this conditional
               Get first Node
               Follow output connections contained in the conditional of each Node
               to Arc, and write string that expresses the direction of the
               control flow
      If Arc atom
         Write Arc class
            Create output file and write header and constructor
            Write Code attribute as the exec( ) method of this class
```

Figure 18. Interpreter pseudocode

CHAPTER V

SAMPLE PROTOCOL IMPLEMENTATION

Graphing The Contract Net Protocol

The contract net (CN) protocol is a high-level protocol for communication among distributed objects [17]. This formalized communication is similar to the common request for contract bids that institutions publish. Following is a textual description that describes the CN.

Contract Net Description

A player in the contract (the initiator) publishes the desire to either buy or sell a commodity. The bids that contractor hopefuls submit for this buying and selling are called "proposals." This call for proposals is often abbreviated as a "cfp."

Once a contractor (responder) realizes the opportunity for business, he examines his resources, and determines whether or not to respond with a proposal. Should the responder provide a proposal, then the cfp/proposal process may continue. The initiator can carry on negotiation conversations with more that one responder. However, wanting to receive the best possible deal, the initiator reveals information only relevant to that responder, and all negotiations are kept confidential. The initiator eventually determines that the proposal is acceptable, or that a solution will not converge in the allotted time for negotiation, and notifies the responder that the negotiation has either succeeded or failed. Once a proposal has been accepted, all other negotiating parties are notified that propos-

als are no longer considered, and that the negotiation has completed. At that point, the initiator provides the responder with the contract.

Formalization of the Description

Using the textual description provided, the next step is to examine the description and determine which parts of the description map directly to Sends and Receives, and which parts map as Actions. At the same time, careful attention must be paid to which atoms will be applied to the initiator, and which to the responder. It is best to list those behaviors of the initiator separate from those of the responder, to facilitate the division. Tables 6 and 7 list all of the possible Send, Receive, and Action behaviors of the initiator and responder roles of the CN.

Table 6. Initiator behavior in the CN

Initiator	
Atom type	**Function**
Send	Cfp Accept
Receive	Propose Refuse
Action	Prepare the cfp Evaluate the proposal • Acceptable • Unacceptable

Table 7. Responder behavior in the CN

Responder	
Atom type	**Function**
Send	Propose Refuse
Receive	Cfp Accept Inform
Action	Evaluate capabilities • Capable • Not capable Prepare the proposal

Graphical Implementation of the Formalization

The high-level portion of the graphical component makeup is complete. The next step is to integrate these components into the modeling environment using the developed paradigm.

For a legend of the modeling concepts used and their representative icons in this paradigm, see Figure 19.

Figure 19. Legend of the parts used when building the model of a protocol.

First, the initiator and responder roles are implemented in Node models. Figure 20 shows the graphical implementation of the initiator role of the CN. Note that some extra message types are allowed to be received in this implementation, emanating from the `t_r_n_p` Receive atom. Those four letter are mnemonics for `timeout`, `refuse`, `not-understood`, and `propose`. The propose connection extends to the EvalProposal icon, while the other three go to the Fail icon. The reason for the additional connections is to allow for the interpretation of exception conditions (which are not present in the textual description at the beginning of this chapter).

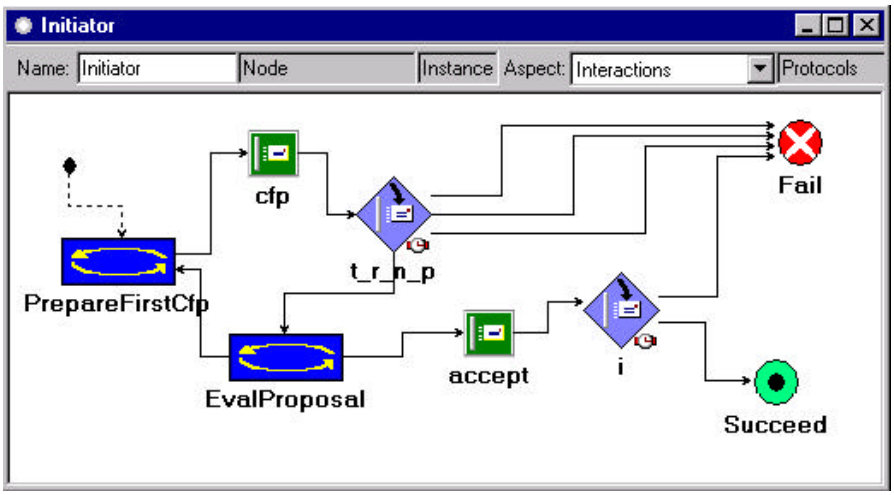


Figure 20. The contract net initiator

Note the "EvalProposal" icon, and that connections go to the "accept" and "PrepareFirstCfp" icons. Those connections have attributes with values that directly correspond to the return values set in Figure 21. When the EvalProposal atom is written to code, then, it will take either the "ACCEPTABLE" or "UNACCEPTABLE" path.

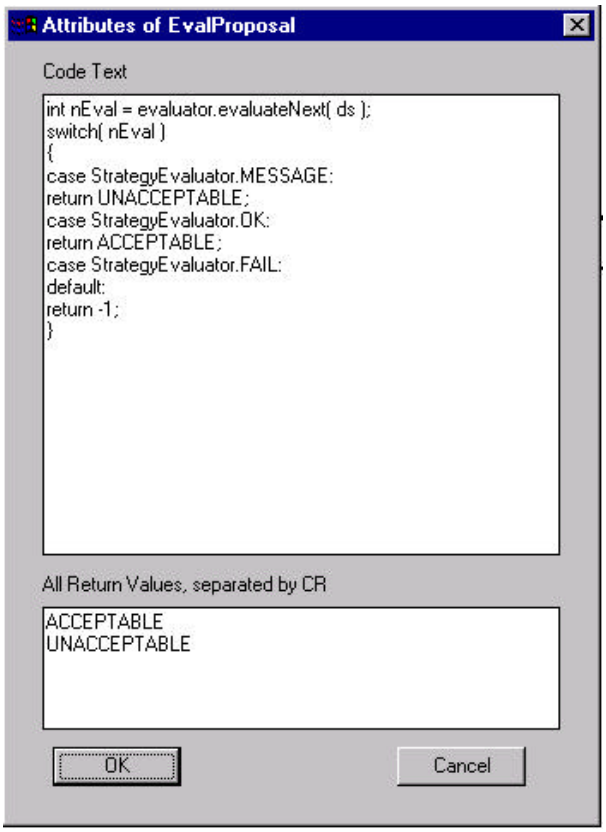
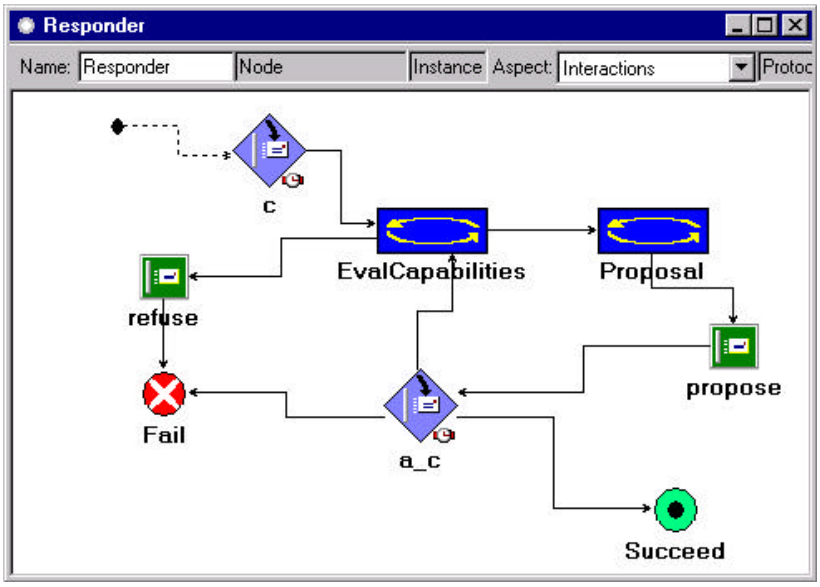


Figure 21. The exit connections of the "EvalProposal" action.

In addition to the possible return values, the "EvalProposal" Action atom also contains the code attribute associated with the behavior of this action. This code segment is compliant with the Zeus API, and a certain degree of familiarity with Zeus is necessary to be proficient in coding the text attribute. However, the Zeus strategy library is accessed with a fairly small amount of code, and the EvalProposal atom produces both necessary conceptual values of the possible outputs of the evaluation of a proposal with only a small amount of specific code.

Each Action atom in both the initiator and responder roles has a similar code fragment (although, some of the fragments are more complex than this one). The re-

sponder role is illustrated in Figure 22, and it reflects the formalized representation of the



responder in Table 7.

Figure 22. The contract net responder

Ignored Atoms

The responder defined in Figure 22 graphically implements a concept that Zeus ignores: the expectation to receive a message before the first send. In Chapter IV it was noted that when Zeus calls a protocol, then a conversation already exists, and the responder must contribute to that conversation before expecting any messages in return. Therefore, the Receive atom named "c" is for conceptualization only; no Receive code is written toward the reception of a message while in this state. Instead, code production begins with "EvalCapabilities."

Other Semantic Inferences

Because of the possibility of many conversations going on at one time between one initiator and many responders, there must be one central location where a decision is made that the negotiation is completed, and that it is time to choose a winner. In Zeus, each conversation is spawned off into a new thread, so that if there are ten responders, then there exist ten threads each participating in the initiator role. When all of these threads complete, then the runtime environment of the initiator chooses a winning responder Agent, and sends the "accept" message to that agent, *not* to its responder thread. Therefore, although conceptually the "accept" message goes to the negotiating party, the negotiation protocol *Node* never receives it.

Since everything after sending an "accept" is outside of the scope of the initiator, once the "accept" message is encountered all atoms are skipped until a Succeed or Fail atom is found. Likewise, since the responder negotiation protocol does not actually send the "inform" message to the initiator, any Sends of performative type "inform" will be skipped.

Completing the Protocol Definition

With the nodes completely defined now, the next step is to completely define the protocol in terms of the Zeus runtime engine, by using the Graph atom. In the Protocol model, Graph Layout aspect, the Nodes should connect to a Graph atom that describes the name of the output protocol.

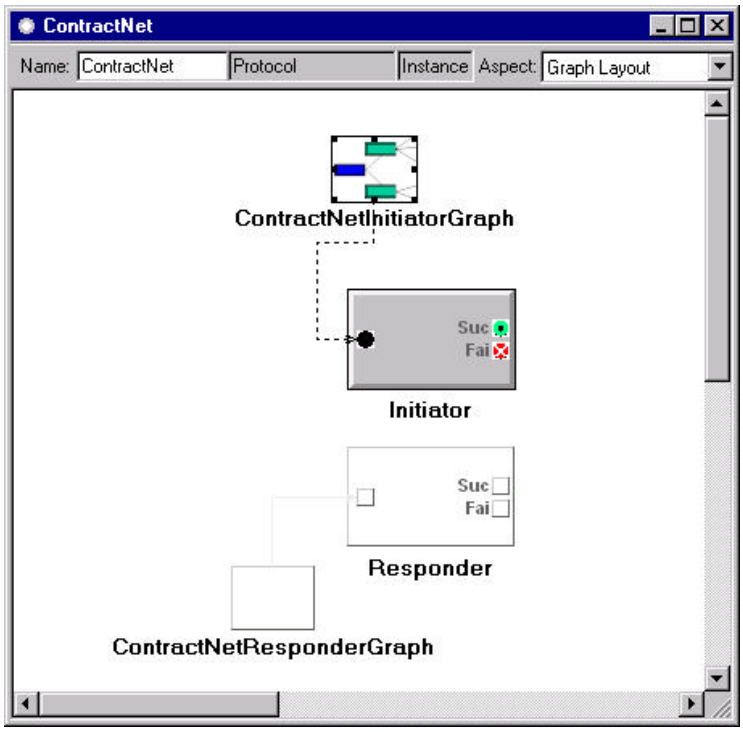


Figure 23. Graph Layout aspect of the Contract Net Protocol model

In Figure 23 the model depicts a Graph atom named "ContractNetInitiatorGraph" that owns a Node called "Initiator" through conditional control. Below, in the same figure, is the corresponding Graph atom for the responder role. Once all of the Node models are owned by the appropriate Graph atom, the specification for the entire negotiation protocol is complete.

Checking Completeness

In order to ensure that the usage of sends and receives by each role in the negotiation is accurate, the State Interconnection aspect of the Protocol model enables the modeler to visually check whether the sends and receives match up, based on the state of the negotiation. Figure 24 shows this aspect of the Protocol model.

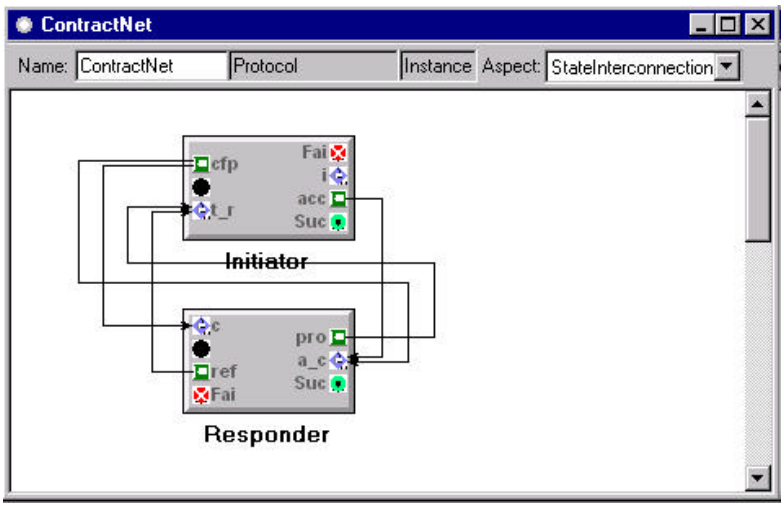


Figure 24. Matching the Sends and Receives in the State Interconnection aspect

There is a Receive atom that does not have a corresponding Send in this case: the "inform" Receive atom in the "Initiator" model. As previously mentioned, however, this atom plays no part in the Zeus output anyway, so the missing correspondence is not an error.

Synthesizing and Examining the Output Implementation

When the model is interpreted, Java source files are produced. One note as to the format of the source files is that the files are written without tabbed logical arrangement. This decision was made based on the observation that incorporating such logic into the interpreter would be an extraordinary amount of work, considering there are several Java environments that provide code formatting as a function of the developing environment. However, the interpreter does not currently format the output, so it is recommended that before examining any output code for correctness that the code be formatted.

The following code segments are a portion of the output file for the initiator Node model.

```
if( msg_type == "propose" )
{
   int int1 = EvalProposal( );
   switch( int1 )
   {
   case ACCEPTABLE:
       ls.result = ds;
       output = ls;
       return OK;
   case UNACCEPTABLE:
       int int2 = PrepareFirstCfp( );
       switch( int2 )
       {
       case DONE:
           engine.continue_dialogue(ls.key,
                                    ls.agent,
                                    "cfp",
                                    evaluator.getGoals( )
                                    );
           return WAIT;
       default:
           return FAIL;
       }
   default:
       return FAIL;
   }
}
```

Figure 25. Output representing connection named “propose”

Figure 25 illustrates the output of one connection of a Receive atom in the “Initiator” Node model. Notice the state machine format through which the logic traces first to “EvalProposal,” then to “PrepareFirstCfp,” and finally to the continuation of the dialog through the sending of a “cfp” message. Figures 26 and 27 show the implementation code of the “EvalProposal” and “PrepareFirstCfp” subroutines, respectively. For a complete listing of all source codes generated by the interpreter for this example, please refer to Appendix B.

```
protected int PrepareFirstCfp(  )
{
    int nEvaluated;
    if( bFirst )
    {
        nEvaluated = evaluator.evaluateFirst( ls.goal,info );
        bFirst = false;
    }
    else
    {
        return DONE;
    }
    switch( nEvaluated )
    {
    case StrategyEvaluator.MESSAGE:
        return DONE;
    default:
        return -1;
    }
}
```

Figure 26. Action code segment, as implemented by the Zeus interpreter

```
protected int EvalProposal(  )
{

    int nEval = evaluator.evaluateNext( ds );
    switch( nEval )
    {
    case StrategyEvaluator.MESSAGE:
        return UNACCEPTABLE;
    case StrategyEvaluator.OK:
        return ACCEPTABLE;
    case StrategyEvaluator.FAIL:
    default:
        return -1;
    }
}
```

Figure 27. Action code segment, as implemented by the Zeus interpreter

Linking to the Agent Runtime Environment

The Zeus runtime environment links to the output file of the Graph atom. For the CN initiator role, this would be the Java file named `ContractNetInitiatorGraph.java`. After the generated Java source files are compiled, then the environment is ready to include the newly implemented interaction protocol.

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

The modeling and program synthesis environment presented in this thesis does three major things.

1. It provides a graphical language for the expression of models.
2. It provides an agent domain specific medium in that graphical environment in which to express interaction protocols. And finally,
3. It translates the graphical models which are in the language of the agent domain into compilable code that the agent runtime system can utilize.

The target user of this modeling environment may come from either extreme of the agent programming world. The experienced agent programmer can use the graphical language to lay out his concept, and then fill in the Action atoms with the implementation code. Then, the graphical model provides a self-documenting explanation of the logic of the protocol, as well as foundation material for understanding the idiosyncrasies of the agent environment for which the protocol was developed (e.g., Zeus does not require certain messages to conclude the negotiation).

The other extreme is the experienced protocol developer who has no experience with agent development. He may use the modeling environment to lay out a rough sketch or sets of ideas of the way in which a protocol should work, perhaps even from the message matching aspect of the Protocol model only, and then pass along the idea to a more experienced programmer to complete the model implementation.

Continuing Research

A more revised implementation of the modeling environment is currently under development. The new DSME is essentially the same as the Node model of the paradigm presented in this thesis. The objective in the newer version was to remove as much superfluous information as possible, and return to the lightweight agent concepts of sending and receiving messages, rather than include information about stringing together protocol nodes and connecting them with a graph object.

The in-production paradigm includes the ability to add together protocols, but it implements that logic by placing the entire logic into one file, rather than separate ones. An interpreter is under development to translate models of the new paradigm into MadKit agents [6].

MadKit is a different type of agent package than Zeus. One advantage of Zeus is that it provides an infrastructure for multiple conversations that includes a message router. This allows an agent to respond to different kinds of messages by starting new threads and executing the coordination graph most appropriate. However MadKit is a simplified agent package that provides methods for sending and receiving messages, but no build in message router to ensure that the correct portions of the agent get the right messages. The solution is to provide a message router that is built on top of the MadKit software through the interpreter. Otherwise, messages could get lost, and the state of the interaction would be out of synchronization.

Once the Zeus interpreter is ported for use with MadKit, the modeling environment will be proven to handle the interpretation of models to multiple agent environments.

Future Work

The modeling environment presented in this thesis, as well as the one under current development, present the implementation of internal agent behavior as a dialog that takes as its parameter agent architecture specific code. Because of the nature of expressing this internal behavior in the language of the specific target environment, the modeling environment *cannot* produce compilable code for two different agent environments from the *same GME project*. This is because in producing the output code, if the same code attributes were written for the Action atoms, then at least one of the agent environments would not compile.

The elegant way to solve this problem is to design an agent interaction layer. This proposed layer would serve as an API between the interpreter and the agent environment. Certain constructs available in all major languages (e.g. string, int, bool) would be available, as well as loop structures (while, for) and if-then-else statements. The most difficult portion of this implementation would be to provide methods for accessing the internal agent structures, such as the ontology, strategy functions, etc. The reason this has not been attempted is due to its complexity. Until a formal agent API is developed by FIPA or some other governing body, this is not an achievable research task, and even then, the model could be said to fully support only that standard.

APPENDIX A

MODELING

This appendix is presented to familiarize readers with the terms and concepts that surround any discussion of modeling. The terms herein described are specific to the Graphical Modeling Environment (GME), developed at Vanderbilt University, and are taken from [15].

Model Integrated Computing (MIC) through the GME provides an environment in which to graphically model domain specific problems. Domain concepts are described in a modeling *paradigm*, which is the definition of the entities and relationships allowed in the given domain. Inside the paradigm are one or more *categories*, which are containers of sets of models, usually in hierarchical form.

The basic modeling objects are *atoms* and *models*. An atom is the elementary object of a model, and is also known as an *atomic part*. An atom is represented with an icon, and may contain a predefined set of attributes, which may be given values by the user at model building time.

A model is a compound object. Models may contain other objects, and may also contain relationships between those objects along with ways to visualize them. Models may contain other models, atoms, *references*, *connections*, and *conditionals*, and may visualize them with *aspects*. All types of objects (models, atoms, references, connections) may have attributes.

Atoms contained in models may be displayed through the model as a *port*. This means that when the model is viewed as a part within another model, then the atoms within that model are viewable.

A reference is similar to a pointer in a programming language, in that it points toward a real object in GME. References may point to atoms, models, or even other references. A port may show up in a model reference, at which time it is known as a *reference port*.

Connections are relationships between atoms, atom references, ports, and reference ports. Only parts which cannot contain other parts may be the ends of a connection.

A conditional is similar to a set. There is a controller (or a set of controllers) to which the set "belongs", and then a set of parts and connections that are members of the conditional. A conditional is a way to associate parts together without cluttering the diagram with connections.

APPENDIX B

CONTRACT NET OUTPUT CODE LISTING

There are four main output files associated with the synthesis of the implementation code of the CN protocol in Zeus. Two Node files (`Initiator.java` and `Responder.java`), and two Graph files (`ContractNetInitiatorGraph.java` and `ContractNetRespondent.java`).

Initiator.java

```
package zeus.isis.ip;

import java.util.*;
import zeus.util.*;
import zeus.concepts.*;
import zeus.actors.*;
import zeus.actors.rtn.*;
import zeus.actors.rtn.util.*;



public class Initiator extends Node {
   protected static final double DELTA_TIME = 0.25;
   public Initiator( ) {
      super( "Initiator" );
   }

   //memory useful for backtracking
   private StrategyEvaluator evaluator = null;
   Engine engine;
   DelegationStruct ds;
   ProtocolDbResult info;
   Goal goal;
   LocalDStruct  ls;
   boolean bFirst = true;


   // Global variables defined as a node attribute {

   //} end of attribute defined globals vars

   //// Return value constants from all subroutines {
   protected final int DONE = 0;
   protected final int ACCEPTABLE = 0;
   protected final int UNACCEPTABLE = 1;
   //// } End

   protected int exec(  )
   {

      engine = context.Engine( );

      ls = (LocalDStruct)input;
      info = ( ProtocolDbResult )ls.any;
```

```
    goal = ( Goal )ls.goal.elementAt( 0 );


    double ct = goal.getConfirmTime( ).getTime( );
    timeout = ct - 1.5*DELTA_TIME;

    Core.DEBUG( 3, getDescription( ) + " Pre-timeout = " + timeout );
    Core.DEBUG( 3, getDescription( ) + " ls.gs.timeout = " + ls.gs.timeout );
    if( ls.gs.timeout != 0 )
    {
       timeout = Math.min( timeout, context.now( ) + ls.gs.timeout );
    }

    Core.DEBUG( 3, getDescription( ) + " Post-timeout = " + timeout );
    Time t = new Time( timeout );
    for( int i = 0; i < ls.goal.size( ); i++ )
    {
       goal = ( Goal )ls.goal.elementAt( i );
       goal.setReplyTime( t );
    }
    msg_wait_key = ls.key;


    evaluator = ( StrategyEvaluator )createObject( info.strategy );
    if( evaluator == null ) return FAIL;
    evaluator.set( context );
    ls.gs.evaluators.add( evaluator );
    evaluator.set( ls.gs.evaluators );
    Core.DEBUG( 3, "About to enter a GME-produced protocol (exec)..." );



    int int0 = PrepareFirstCfp( );
    switch( int0 )
    {
    case DONE:
       engine.new_dialogue(ls.key,
                           ls.agent,
                           "cfp",
                           evaluator.getGoals( )
                           );
       return WAIT;
    default:
       return FAIL;
    }

}

protected int continue_exec(  )
{

    Core.DEBUG( 2, "Initiator continue_exec" );

    if( context.now( ) > timeout )
    {
       Core.DEBUG( 2, "Initiator Fail: " + context.now( ) + " > " + timeout );
       return FAIL;
    }

    ds = engine.replyReceived( ls.key );
    String msg_type = ds.msg_type;
    engine = context.Engine( );

    ls = (LocalDStruct)input;
    Core.DEBUG( 3, "About to enter a GME-produced protocol (continue_exec)..." );


    if( msg_type == "propose" )
    {
       int int1 = EvalProposal( );
       switch( int1 )
```

```
        {
        case ACCEPTABLE:
            ls.result = ds;
            output = ls;
            return OK;
        case UNACCEPTABLE:
            int int2 = PrepareFirstCfp( );
            switch( int2 )
            {
            case DONE:
                engine.continue_dialogue(ls.key,
                                         ls.agent,
                                         "cfp",
                                         evaluator.getGoals( )
                                         );
                return WAIT;
            default:
                return FAIL;
            }
        default:
            return FAIL;
        }
    }
    else if( msg_type == "timeout" )
    {
        return FAIL;
    }
    else if( msg_type == "refuse" )
    {
        return FAIL;
    }
    else if( msg_type == "not_understood" )
    {
        return FAIL;
    }
    if( msg_type == "timeout" )
    {
        return FAIL;
    }
    else if( msg_type == "inform" )
    {
        ls.result = ds;
        output = ls;
        return OK;
    }
    // should never get here
    return FAIL;

}

protected int PrepareFirstCfp(  )
{

    int nEvaluated;
    if( bFirst )
    {
        nEvaluated = evaluator.evaluateFirst( ls.goal,info );
        bFirst = false;
    }
    else
    {
        return DONE;
    }
    switch( nEvaluated )
    {
    case StrategyEvaluator.MESSAGE:
        return DONE;
    default:
        return -1;
    }
}
```

```
    protected int EvalProposal(  )
    {

        int nEval = evaluator.evaluateNext( ds );
        switch( nEval )
        {
        case StrategyEvaluator.MESSAGE:
            return UNACCEPTABLE;
        case StrategyEvaluator.OK:
            return ACCEPTABLE;
        case StrategyEvaluator.FAIL:
        default:
            return -1;
        }
    }


}
```

```
package zeus.isis.ip;

import java.util.*;
import zeus.util.*;
import zeus.concepts.*;
import zeus.actors.*;
import zeus.actors.rtn.*;
import zeus.actors.rtn.util.*;



public class Responder extends Node {
   protected static final double DELTA_TIME = 0.25;
   public Responder( ) {
      super( "Responder" );
   }

   //memory useful for backtracking
   private StrategyEvaluator evaluator = null;
   Engine engine;
   DelegationStruct ds;
   ProtocolDbResult info;
   Goal goal;
   GraphStructgs;
   boolean bFirst = true;


   // Global variables defined as a node attribute {

   //} end of attribute defined globals vars

   //// Return value constants from all subroutines {
   protected final int IS_CAPABLE = 0;
   protected final int NOT_CAPABLE = 1;
   protected final int DONE = 0;
   //// } End

   protected int exec(  )
   {

      engine = context.Engine( );

      gs = (GraphStruct)input;
      info = ( ProtocolDbResult )gs.any;
      goal = ( Goal )gs.goal.elementAt( 0 );


      evaluator = ( StrategyEvaluator )createObject( info.strategy );
      if( evaluator == null ) return FAIL;
      evaluator.set( context );
      gs.evaluators.add( evaluator );
      evaluator.set( gs.evaluators );
      Core.DEBUG( 3, "About to enter a GME-produced protocol (exec)..." );



      timeout = goal.getConfirmTime().getTime();
      msg_wait_key = gs.key;


      int int0 = EvalCapabilities( );
      switch( int0 )
      {
      case NOT_CAPABLE:
         engine.continue_dialogue(gs.key,
                                  gs.agent,
                                  "refuse",
```

```
                            evaluator.getGoals( )
                            );
        return FAIL;
    case IS_CAPABLE:
        int int1 = Proposal( );
        switch( int1 )
        {
        case DONE:
            engine.continue_dialogue(gs.key,
                                gs.agent,
                                "propose",
                                evaluator.getGoals( )
                                );
            return WAIT;
        default:
            return FAIL;
        }
    default:
        return FAIL;
    }

}

protected int continue_exec(  )
{

    Core.DEBUG( 2, "Responder continue_exec" );

    if( context.now( ) > timeout )
    {
        Core.DEBUG( 2, "Responder Fail: " + context.now( ) + " > " + timeout );
        return FAIL;
    }

    ds = engine.replyReceived( gs.key );
    String msg_type = ds.msg_type;
    engine = context.Engine( );

    gs = (GraphStruct)input;
    Core.DEBUG( 3, "About to enter a GME-produced protocol (continue_exec)..." );


    if( msg_type == "cfp" )
    {
        int int2 = EvalCapabilities( );
        switch( int2 )
        {
        case NOT_CAPABLE:
            engine.continue_dialogue(gs.key,
                                gs.agent,
                                "refuse",
                                evaluator.getGoals( )
                                );
            return FAIL;
        case IS_CAPABLE:
            int int3 = Proposal( );
            switch( int3 )
            {
            case DONE:
                engine.continue_dialogue(gs.key,
                                    gs.agent,
                                    "propose",
                                    evaluator.getGoals( )
                                    );
                return WAIT;
            default:
                return FAIL;
            }
        default:
            return FAIL;
        }
```

```
        }
        else if( msg_type == "timeout" )
        {
            return FAIL;
        }
        else if( msg_type == "accept-proposal" )
        {
            gs.confirmed = true;
            gs.confirmed_goal = ds.goals;
            output = gs;
            return OK;
        }
        // should never get here
        return FAIL;

    }

    protected int EvalCapabilities(  )
    {

        return IS_CAPABLE;
    }

    protected int Proposal(  )
    {

        int nEvaluated;
        if( bFirst )
        {
            nEvaluated = evaluator.evaluateFirst( gs.goal, info );
            bFirst = false;
        }
        else
        {
            nEvaluated = evaluator.evaluateNext( ds );
        }
        switch( nEvaluated )
        {
        case StrategyEvaluator.MESSAGE:
            return DONE;
        default:
            return -1;
        }
    }


}
```

```
/*
* The contents of this file are subject to the BT "ZEUS" Open Source
* Licence (L77741), Version 1.0 (the "Licence"); you may not use this file
* except in compliance with the Licence. You may obtain a copy of the Licence
* from $ZEUS_INSTALL/licence.html or alternatively from
* http://www.labs.bt.com/projects/agents/zeus/licence.htm
*
* Except as stated in Clause 7 of the Licence, software distributed under the
* Licence is distributed WITHOUT WARRANTY OF ANY KIND, either express or
* implied. See the Licence for the specific language governing rights and
* limitations under the Licence.
*
* The Original Code is within the package zeus.*.
* The Initial Developer of the Original Code is British Telecommunications
* public limited company, whose registered office is at 81 Newgate Street,
* London, EC1A 7AJ, England. Portions created by British Telecommunications
* public limited company are Copyright 1996-9. All Rights Reserved.
*
* THIS NOTICE MUST BE INCLUDED ON ANY COPY OF THIS FILE
*/
// This is an automatically generated file from the GME.
// This is file 'ContractNetInitiatorGraph.java'
package zeus.isis.ip;

import java.util.*;
import zeus.actors.rtn.*;
import zeus.actors.rtn.util.*;
import zeus.util.*;

public class ContractNetInitiatorGraph extends Graph {


    private static final String[][] entry = {
        {"zeus.isis.ip.Initiator"}
    };
    public ContractNetInitiatorGraph() {
        super("ContractNetInitiatorGraph",entry,"zeus.isis.ip.Initiator");
    }

}
```

```
/*
* The contents of this file are subject to the BT "ZEUS" Open Source
* Licence (L77741), Version 1.0 (the "Licence"); you may not use this file
* except in compliance with the Licence. You may obtain a copy of the Licence
* from $ZEUS_INSTALL/licence.html or alternatively from
* http://www.labs.bt.com/projects/agents/zeus/licence.htm
*
* Except as stated in Clause 7 of the Licence, software distributed under the
* Licence is distributed WITHOUT WARRANTY OF ANY KIND, either express or
* implied. See the Licence for the specific language governing rights and
* limitations under the Licence.
*
* The Original Code is within the package zeus.*.
* The Initial Developer of the Original Code is British Telecommunications
* public limited company, whose registered office is at 81 Newgate Street,
* London, EC1A 7AJ, England. Portions created by British Telecommunications
* public limited company are Copyright 1996-9. All Rights Reserved.
*
* THIS NOTICE MUST BE INCLUDED ON ANY COPY OF THIS FILE
*/
// This is an automatically generated file from the GME.
// This is file 'ContractNetResponderGraph.java'
package zeus.isis.ip;

 import java.util.*;
import zeus.actors.rtn.*;
import zeus.actors.rtn.util.*;
import zeus.util.*;

public class ContractNetResponderGraph extends Graph {


    private static final String[][] entry = {
        {"zeus.isis.ip.Responder"}
    };
     public ContractNetResponderGraph() {
        super("ContractNetResponderGraph",entry,"zeus.isis.ip.Responder");
    }

}
```

APPENDIX C

PARADIGM SPECIFICATION

The paradigm that defines the DSME used in this thesis is described by the protocol.edf file. The file was created using the metamodeling environment in use at Vanderbilt University. The UML class diagram that was used in the metamodeling environment is in Figure 28. The complete paradigm is displayed in the printout of the file `protocol.edf`, and the constraints are described using the MultiGraph Constraint Language (MCL) in the file `protocol.mcl`.

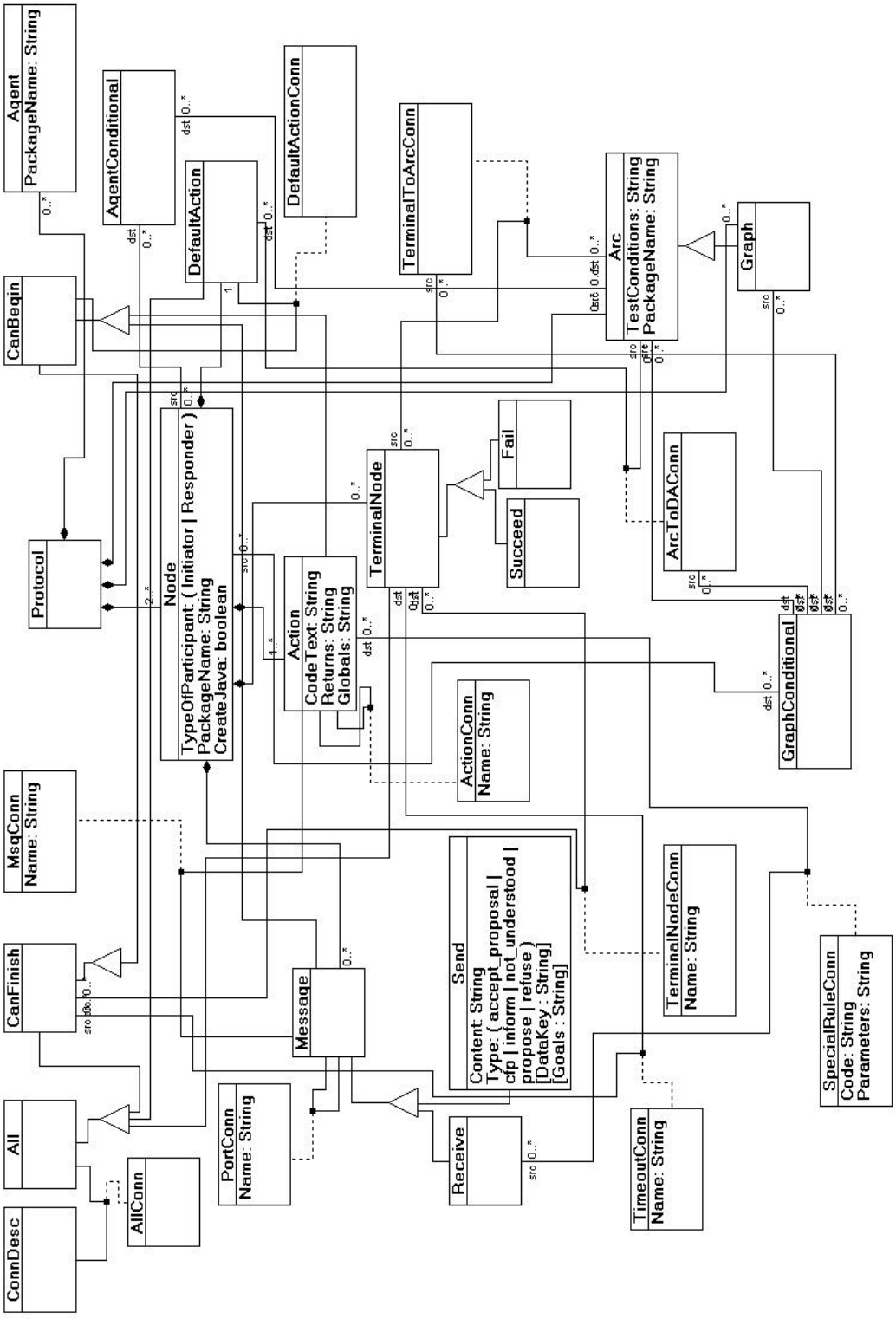


Figure 28. Complete UML Class diagram

```
//
// << Generated EDF (Protocol.edf) >>
// Date: Wednesday, March 01, 2000
// Time: 14:04:58
// Meta interpreter information:
//   Version: 1
//   Build: 51
//   Date: 01.06.00
//

paradigm Protocol;

const {
Protocol,Code,Returns,Action,Content,
ACCEPT,CFP,INFORM,NOT_UNDERSTOOD,PROPOSE,
REFUSE,MessageType,DataKey,Goals,Send,
Receive,DefaultAction,Succeed,Fail,ConnDesc,
PackageName,Arc,Agent,Graph,ConnectionName,
PortConn,Node,Participants,ArcToDAConn,TerminalToArcConn,
AgentConditional,GraphConditional,GraphLayout,INIT,RESP,
TypeParticipantMenu,Globals,CreateJava,MsgConn,DefaultActionConn,
TerminalNodeConn,Parameters,ActionConn,AllConn,SpecialRuleConn,
TimeoutConn,Protocols}

atom Action "Action" {
  icon "action.bmp";
  attrs {
    Code : page "Code Text" (15 30) "";
    Returns : page "All Return Values, separated by CR" (4 30) "";
  }
}
atom Send "Send" {
  icon "send_msg.bmp";
  attrs {
    Content : field  "Message content: " "";
    MessageType : menu "Message Type"
      {
        "Accept-Proposal" ACCEPT default;
        "cfp" CFP;
        "Inform" INFORM;
        "Not Understood" NOT_UNDERSTOOD;
        "Propose" PROPOSE;
        "Refuse" REFUSE;
      };
    DataKey : field  "Data Key" "";
    Goals : field  "Goals" "";
  }
}
atom Receive "Receive" {
  icon "rcv_msg.bmp";
}
atom DefaultAction "DefaultAction" {
  icon "default.bmp";
}
atom Succeed "Succeed" {
  icon "succeed.bmp";
}
atom Fail "Fail" {
  icon "fail.bmp";
}
atom ConnDesc "ConnDesc" {
  icon "conndesc.bmp";
}
atom Arc "Arc" {
  icon "arc.bmp";
  attrs {
    PackageName : field  "Java Package" "";
    Code : page "Code Text" (15 30) "";
```

```
  }
}
atom Agent "Agent" {
  icon "agent.bmp";
}
atom Graph "Graph" {
  icon "graph.bmp";
  attrs {
    PackageName : field  "Java Package" "";
    Code : page "Code Text" (15 30) "";
  }
}
model Protocol "Protocol" {
 Participants "StateInterconnection" {
  conns {
    PortConn { 1 solid line arrow } :
      { Node Receive -> Node Receive }
      { Node Receive -> Node Send }
      { Node Send -> Node Receive }
      { Node Send -> Node Send }
      attrs {
        ConnectionName : field  "Name of Connection" "";
      };
  }
  parts {
    Agent : Agent;
    Node : Node;
  }
 }
 GraphLayout "Graph Layout" {
  conns {
    ArcToDAConn { 1 dash1_1 line arrow } :
      { Arc  -> Node DefaultAction }
      { Graph  -> Node DefaultAction };
    TerminalToArcConn { 1 dash1_1 line arrow } :
      { Node Fail -> Arc  }
      { Node Fail -> Graph  }
      { Node Succeed -> Arc  }
      { Node Succeed -> Graph  };
  }
  conds {
    AgentConditional Agent:
      { }
      { Node Arc };
    GraphConditional Graph:
      { TerminalToArcConn ArcToDAConn }
      { Node Arc Graph };
  }
  parts {
    Agent : Agent inherited;
    Arc : Arc;
    Graph : Graph;
    Node : Node inherited;
  }
 }
}
model Node "Node" {
 Participants "Interactions" {
  attrs {
    PackageName : field  "Java Package" "";
    TypeParticipantMenu : menu "Participant Type"
      {
        "Initiator" INIT default;
        "Respondent" RESP;
      };
    Globals : page "Global Variables" (4 30) "";
    CreateJava : toggle "Create Java File" true;
  }
  conns {
    MsgConn { 1 solid line arrow } :
      { Action  -> Send  }
```

```
    { Send  -> Action  }
    { Action  -> Receive  }
    { Receive  -> Action  }
    attrs {
      ConnectionName : field  "Name of Connection" "";
    };
  PortConn { 1 solid line arrow } :
    { Send  -> Send  }
    { Send  -> Receive  }
    { Receive  -> Send  }
    { Receive  -> Receive  }
    attrs {
      ConnectionName : field  "Name of Connection" "";
    };
  DefaultActionConn { 1 dash1_1 line arrow } :
    { DefaultAction  -> Send  }
    { Send  -> DefaultAction  }
    { DefaultAction  -> Receive  }
    { Receive  -> DefaultAction  }
    { DefaultAction  -> Action  }
    { Action  -> DefaultAction  };
  TerminalNodeConn { 1 solid line arrow } :
    { Send  -> Succeed  }
    { Send  -> Fail  }
    { Receive  -> Succeed  }
    { Receive  -> Fail  }
    { Action  -> Succeed  }
    { Action  -> Fail  }
    attrs {
      ConnectionName : field  "Name of Connection" "";
    };
  ActionConn { 1 solid line arrow } :
    { Action  -> Action  }
    attrs {
      ConnectionName : field  "Name of Connection" "";

      Code : page "Code Text" (15 30) "";

      Parameters : page "Parameters" (4 30) "";
    };
  AllConn { 1 solid line arrow } :
    { Send  -> ConnDesc  }
    { ConnDesc  -> Send  }
    { Receive  -> ConnDesc  }
    { ConnDesc  -> Receive  }
    { Action  -> ConnDesc  }
    { ConnDesc  -> Action  }
    { DefaultAction  -> ConnDesc  }
    { ConnDesc  -> DefaultAction  }
    { Succeed  -> ConnDesc  }
    { ConnDesc  -> Succeed  }
    { Fail  -> ConnDesc  }
    { ConnDesc  -> Fail  };
  SpecialRuleConn { 1 dash1_1 line arrow } :
    { Receive  -> Action  }
    attrs {
      Code : page "Code Text" (15 30) "";

      Parameters : page "Parameters" (4 30) "";
    };
  TimeoutConn { 1 dash1_1 line arrow } :
    { Send  -> Succeed  }
    { Send  -> Fail  }
    { Receive  -> Succeed  }
    { Receive  -> Fail  }
    { Action  -> Succeed  }
    { Action  -> Fail  }
    attrs {
      ConnectionName : field  "Name of Connection" "";
    };
}
```

```
  parts {
    Send : Send link;
    Receive : Receive link;
    Action : Action;
    DefaultAction : DefaultAction link;
    Succeed : Succeed link;
    Fail : Fail link;
    ConnDesc : ConnDesc;
  }
 }
 GraphLayout "Graph Layout" {
  parts {
    Fail : Fail link inherited;
    Succeed : Succeed link inherited;
    DefaultAction : DefaultAction link inherited;
  }
 }
}
category Protocols "CategoryPart" { Node Protocol }
```

```
//
// Generated MCL (Protocol.mcl)
// Date: Thursday, July 20, 2000
// Time: 10:11:40
//

constraint ArcFromConstraint()
priority=0
"An arc must have exactly one 'From' connection." {
  parts( "Arc" )->forAll( a2 | a2.connectedFrom( )->size( ) = 1 )
}

on (connect_event, create_event, close_event, delete_event)
constraint SendOutputConstraint()
priority=0
"A send must have one and only one output connection" {
  models( "Node" )->forAll( n | n.parts( "Send" )->connectionsTo( )->size( ) = 1 )
}

constraint SynthesizedConstraint0()
priority=0
"Every Protocol-kind object must contain 2..* Node-kind object(s)" {
  models()->select(m|m.kindOf()="Protocol")->forAll(model |
   (model.parts()->select(p | p.kindOf()="Node")->size() >= 2))
}

constraint SynthesizedConstraint1()
priority=0
"Every Node-kind object must contain 1 DefaultAction-kind object(s)" {
  models()->select(m|m.kindOf()="Node")->forAll(model |
   (model.parts()->select(p | p.kindOf()="DefaultAction")->size() = 1))
}
```

REFERENCES

[1] S. Franklin, A. Graesser, "Is It an Agent, Or Just a Program? A Taxonomy for Autonomous Agents," *Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages*, Springer-Verlag, 1996.

[2] N. R. Jennings, M. Wooldridge, "Software Agents", *IEE Review*, pp. 17-20, Jan. 1996.

[3] H. Nwana, "Software Agents: An Overview," *Knowledge Engineering Review Journal*, Vol. 11, No. 3, pp. 205-234, Nov. 1996.

[4] M. Barbuceanu, M. S. Fox, "Capturing and Modeling Coordination Knowledge for Multi-Agent Systems," *International Journal of Cooperative Information Systems*, Vol 5, No. 2, pp. 275-314, 1996.

[5] Foundation for Intelligent Physical Agents, "FIPA 97 Specification," Part 1, Ver. 2.0, Oct. 1998.

[6] O. Gutknecht, J. Ferber, F. Michel, "The MadKit Agent Platform Architecture," *Rapport de Recherche, Universite Montpellier*, R.R.LIRMM 000xx, May 2000.

[7] G. Karsai, F. DeCaria, "Model-Integrated On-line Problem-Solving Environment for Chemical Engineering," *IFAC Control Engineering Practice*, Vol. 5, No. 5, pp. 1-9, 1997.

[8] G. Karsai, J. Sztipanovits, S. Padalkar, C. Biegl, "Model Based Intelligent Process Control for Cogenerator Plants," *Journal of Parallel and Distributed Systems*, pp. 90-103, 1992.

[9] E. Long, A. Misra, J. Sztipanovits, "Increasing Productivity at Saturn," *IEEE Computer Magazine*, August, 1998.

[10] A. Misra, G. Karsai, J. Sztipanovits, "Model-Integrated Development of Complex Applications," *Proceedings of the Fifth International Symposium on Assessment of Software Tools*, pp. 14-23, Pittsburgh, PA, June, 1997.

[11] J. Sztipanovits, G. Karsai, "Model-Integrated Computing," *IEEE Computer*, pp. 110-112, April, 1997.

[12] T. Finin, et al., "Specification of the KQML Agent Communication Language," The DARPA Knowledge Sharing Initiative, External Interfaces Working Group, 1992.

[13] L. Bölöni, D. C. Marinescu, "A Multi-Plane State Machine Agent Model," *Fourth International Conference on Autonomous Agents*, Jun. 1999.

[14] J. Collis, D. Ndumu, H. Nwana, L. Lee, "The Zeus Agent Building Tool-Kit," BT Technology Journal, Vol. 16, No. 3, pp. 60-68, Jul. 1998.

[15] A. Ledeczi, M. Maroti, G. Karsai, G. Nordstrom, "Metaprogrammable Toolkit for Model-Integrated Computing," *Proceedings of the Engineering of Computer Based Systems (ECBS) Conference*, pp. 311-317, Nashville, TN, March, 1999.

[16] J. Wakerly, *Digital Design Principles and Practices*, 2nd edition, p. 468, Prentice Hall, 1994.

[17] R. 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, Dec. 1980.

[18] Y. Tahara, A. Ohsuga, S. Honiden, "Agent System Development Method Based on Agent Patterns", *Proceedings of the 21st International Conference on Software Engineering*, ACM Press, pp.356-367, 1999.

ELECTRICAL ENGINEERING

MODEL INTEGRATED PROGRAM SYNTHESIS OF AGENT INTERACTION PROTOCOLS

JONATHAN M. SPRINKLE

Thesis under the direction of Dr. Gabor Karsai

Agent based technology is an approach to distributed computing that employs distributed entities (or agents) to work towards a goal. These agents are the actors in a Multi-Agent System (MAS), and often communicate directly with each other, and not through a general controller. Communication standards define Agent Communication Languages (ACL's), and within an ACL a small set of speech acts (or performatives) are allowed. Two agent systems developed independently may communicate with each other if they both use the same standard ACL. The advantage of communication within agents is that by sharing data through messages, some emergent behavior may occur that allows a MAS to solve a problem through negotiation. Negotiation is the relaxation of constraints, based on the state of the agent interaction.

Agent developers generally define interaction protocols for negotiation using the same language in which the agent itself is implemented. Since most of these languages are low-level textual based languages such as Java or C/C++, the developer is faced with a coding intensive task, not to mention the need to think on the level of the programming language, as well as the level of the final implementation. Also, if the interaction proto-

col, once developed for an agent package, were desired for a different agent package, it would have to be implemented again from scratch.

This thesis uses Model-Integrated Program Synthesis (MIPS) to allow the agent developer to graphically model agent interaction protocols, and produce from that graphical model low-level output code. Since the modeling takes place on a high level, it would also be possible to produce protocols for more than one agent environment using the same modeling environment. Furthermore, the graphical nature of the description of the protocol allows the user to analyze the structure of the interaction protocol as a graph.

Approved_____ Date_____