The 2000 International Conference on Artificial Intelligence (IC-AI'2000)
June 26 - 29, 2000 Monte Carlo Resort, Las Vegas, Nevada

Verification of Agent Behavioral Models[♦]

Timothy H. Lacey and Scott A. DeLoach

Air Force Institute of Technology
Graduate School of Engineering and Management
Department of Electrical and Computer Engineering
Wright-Patterson Air Force Base, OH 45433-7765

Phone: (937) 255-3636 x4622 Fax: (937) 656-4055
timothy.lacey@afit.af.mil, scott.deloach@afit.af.mil

Keywords: multiagent, behavior, concurrent, verification

Abstract

Intelligent software agents are becoming very popular. They are ideal for solving distributed problems that are too difficult for a non-distributed system to solve. Distributed agents can be used to retrieve, filter, and summarize information as well as provide intelligent user interfaces. However, multiagent systems are very complicated to build and must be dependable. Agent conversation protocols, a series of messages passed between agents, are the cornerstone of multiagent systems. Agents also have tasks associated with them that specify how an agent behaves. This paper introduces a formal methodology that automatically verifies the interaction between agents. Agent behavioral specifications are created graphically in the agentTool multiagent development environment. This graphical representation is then transformed into a formal modeling language called Promela that is analyzed by Spin to ensure the interaction between agents is correct. This type of verification provides the user with another tool to ensure the system will perform as expected.

Introduction

Intelligent software agents are becoming very popular. They are ideal for solving distributed problems that are too difficult for a non-distributed system to solve. Distributed agents can be used to retrieve, filter, and summarize information as well as provide intelligent user interfaces. However, multiagent systems are very complicated to build and must be dependable.

Software agents operate in various distributed systems. Open agent systems are those where agents can interact with each other via autonomous and unstructured conversations. Agents may have goals and pursue them with whatever means they have available. Much of the software agent research is targeted for open systems. Closed agent systems are those where agents interact with each other via structured and predictable communication protocols or conversations. All players in the system are known and all conversations follow specific patterns. Military applications and electronic commerce are just two areas where closed multiagent systems are used.

Before a multiagent system can be trusted to perform as expected, the communication protocols between the agents must be formally verified. For example, errors in conversation protocols can prevent orders from getting through to subordinates or financial transactions from being completed. At a higher level of abstraction, agent behavior and interactions can be modeled as concurrent "tasks". Agent tasks are modeled using state transition diagrams and essentially define the behavior of an agent. The verification process includes checking that behavioral models of interacting agents respond to agent messages correctly. This paper introduces a formal methodology that automatically verifies the interaction between agents. Agent behavioral specifications are created graphically in the agentTool multiagent development environment. This graphical representation is then transformed into a formal modeling language called Promela that is analyzed by Spin to ensure the interaction between agents is correct. This type of verification provides the user with another tool to ensure the system will perform as expected.

Background

The best way for software developers to tackle complex, large, or unpredictable domains is by breaking the problem into smaller, more manageable tasks. Software agents can be used to solve these small tasks while working together to solve larger problems. Sub-problems force agents to communicate with each other while working together on the "big picture." Sycara has observed that agents must often operate concurrently in a distributed environment to accomplish a given task (Sycara, 1998).

agentTool

We are currently developing a software development environment, called agentTool, to address the need for a

[♦] The views expressed in this article are those of the authors and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the US Government.

user friendly, robust tool for building multiagent systems. The tool is an integrated environment that allows a user to graphically design a multiagent system, verify the agent conversations with an automated verification tool, and automatically generate the source code for the designed system. The agentTool environment incorporates the Multiagent System Engineering (MaSE) methodology (DeLoach, 1999). MaSE is both a methodology and a language for designing multiagent systems and includes four levels of design: domain, agent, component, and system.

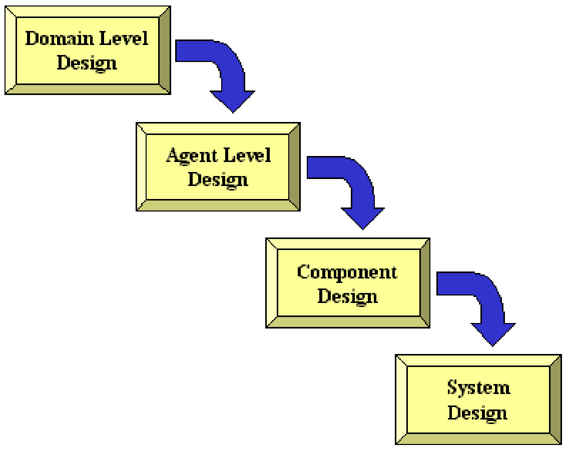

Figure 1: MASE Overview

Figure 1 shows an overview of the MASE methodology. During domain level design, agent classes and agent interactions are defined. During agent level design, the internal architecture of each agent is designed. During component level design, individual components are designed within each agent. Finally, during system level design, we determine which domain agents to use, where they reside, and what data they have access to.

Agents communicate with each other using patterns of messages called *conversations* (Greaves, 1999). Conversations are modeled using state transition diagrams (Pressman, 1997). Given a set of conversation state transition diagrams, communication between agents can be simulated and every possible message combination exercised. Using this approach, conversations are deemed *valid* if the desired message sequence takes place between the communicating agents. This process of deeming the conversations valid or invalid is called *verifying* the agent conversations. Conversations can be verified manually by a human analyst or automatically by intelligent software and automated tools (Lacey, 2000).

Individual agent behavior is described by specifying a concurrent task that is associated with an agent. Modeling an agent's behavior in this manner is very similar to the way we model agent conversations. However, tasks differ from conversations in that conversations are a lower level of detail than tasks. An agent may be composed of one or more tasks that model all of the interactions an agent must implement. Conversations are simply the preferred method of implementing agent-to-agent interaction.

Promela/Spin

We use Promela and Spin to formally model agent interactions (Lacey, 2000). With Promela and Spin, we can detect deadlock, livelock, assertion violations, and many other communication centric errors while efficiently using computer resources.

Automatic Verification of Agent Behavioral Models

The present method of protocol verification requires a human to manually model a protocol in a formal language so the verifier can be used. Formal methods are very difficult to understand and use in this manner. The challenge then is to automatically generate the formal representation of a behavioral model and then use an automated tool to verify that this representation is free from interaction errors. Figure 2 is a top-level view of the overall process.

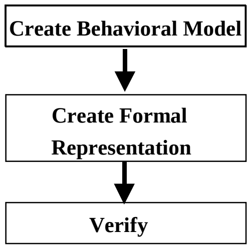

Figure 2: Top Level View of Methodology

Specifying Agent Behavior

In agentTool, the first step in creating an agent is to define an agent role. We model agent behavior using *tasks*, which specifies a single thread of control that defines a single task that the agent can perform. These tasks are specified graphically using state transition diagrams, as shown in Figure 3. All tasks are assumed to start execution upon startup of the agent and continue until the agent terminates or an end state is reached.

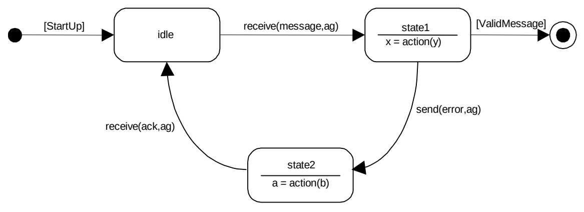

Figure 3: Concurrent Task

We define the entirety of an agent's behavior as consisting of *n concurrent tasks*, which execute in parallel. Actions are used to specify the actual functions carried out by the agent and are performed inside the task states. While these tasks execute concurrently and carry out highlevel behavior, they can be coordinated using internal events. Internal events are passed from one task to another and are specified on the transitions between states. To communicate with other agents, external messages can be sent and received. These messages are specified internally as *send* and *receive* events. These events send and retrieve messages from the message-handling component of the agent, which is assumed to exist. Besides communication with other agents, tasks can interact with the environment via reading percepts or performing operations that affect the environment. This interaction is typically captured via functions defined in the states. By including reasoning within tasks, agents are not "hardwired" or purely reflexive. They can plan, search, or use knowledge-based reasoning to decide on appropriate actions. Concurrent task diagrams allow the modeling of sophisticated coordination protocols such as the Contract Net protocol, the English-Auction protocol, or the Dutch-Auction protocol.

Syntax

The syntax of a concurrent task has two components: states and transitions. As defined above, the states and transitions are similar to the states and transitions of most other finite state models. States represent processing that goes on internal to the agent. This processing is denoted by a sequence of actions. Transitions denote communication between agents or between tasks.

In agentTool, we allow interaction between multiple tasks. Agent behavior models move through various states until eventually, both sides of the interaction end up in valid "end" states and the interaction is complete. The state transition diagram allows us to visualize the various states that a behavioral model goes through and the events that cause the task to move from state to state.

Example

This paper demonstrates how the Contract Net protocol is modeled and verified using agent tasks. Figures 4 and 5 show abstract behavioral models for a "Manager" agent and a "Bidder" agent respectively. The abstract behavioral task model is created using the syntax and semantics described in (DeLoach, 2000). We briefly describe some of the semantics here. The abstract behavioral model is then translated, with human assistance, to a concrete behavior model from which Promela code can be derived. After creating the Promela model, we verify the behavior models are free from undesirable communication properties such as deadlock.

Creating an Abstract Behavioral Model

The beginning state in a behavior model is the *start* state, signified by a solid circle. Any state can be a valid ending state, provided the state has been designated as a valid *end* state. Each state, other than the *start* state, is drawn as an unfilled rounded edge rectangle. The state's name is inside the rectangle. Arrows between states indicate transitions between those states and the direction of the transition. Labels on the arrows indicate the events and actions that take place to cause a transition from one state to another and follow the notation *Trigger [guard condition] transmission(s)*.

The transition label may contain some or all of this information. Each state may have more than one entry point and exit point. If a state has more than one *enabled* exit point, then a priority hierarchy determines which transition is executed. This hierarchy is *received events* (by order of receipt), *send events* (by order of receipt), *received messages* (by order of receipt), *send messages* (by order of receipt), *guard conditions* (multiple guard conditions must be mutually exclusive), and *null transitions* (only one per state).

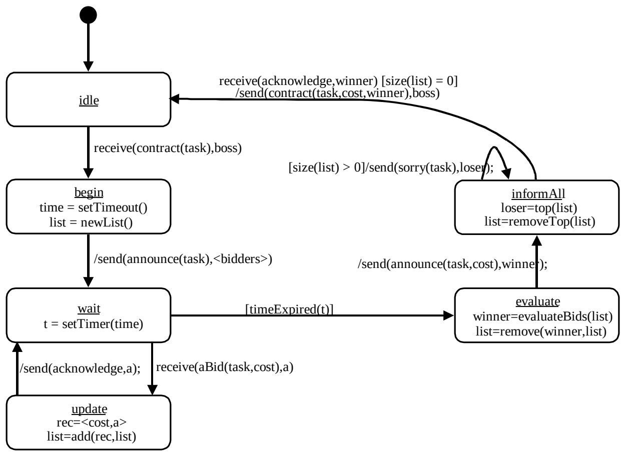
idle
receive(acknowledge,winner) [size(list) = 0]
/send(contract(task,cost,winner),boss)
receive(contract(task),boss)
[size(list) > 0]/send(sorry(task),loser);
begin
time = setTimeout()
list = newList()
informAll
loser=top(list)
list=removeTop(list)
/send(announce(task),<bidders>)
/send(announce(task,cost),winner);
wait
t = setTimer(time)
[timeExpired(t)]
evaluate
winner=evaluateBids(list)
list=remove(winner,list)
/send(acknowledge,a);
receive(aBid(task,cost),a)
update
rec=<cost,a>
list=add(rec,list)

Figure 4: Contract Net Manager Abstract Task

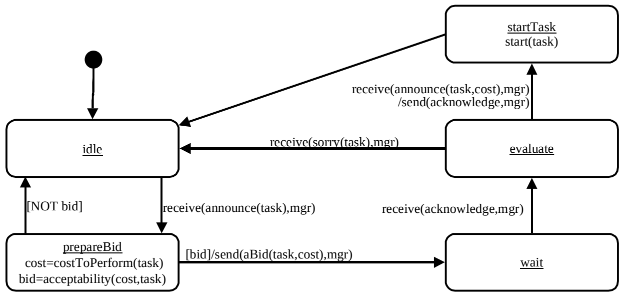

Figure 5: Contract Net Bidder Abstract Task

There is also a third agent task involved in this protocol. A "Boss" agent starts the whole process by sending a *contract* to the manager. Figure 6 shows the abstract Boss agent task.

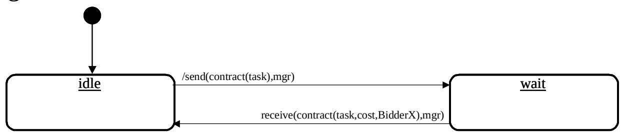
idle
/send(contract(task),mgr)
wait
receive(contract(task,cost,BidderX),mgr)

Figure 6: Contract Net Boss Abstract Task

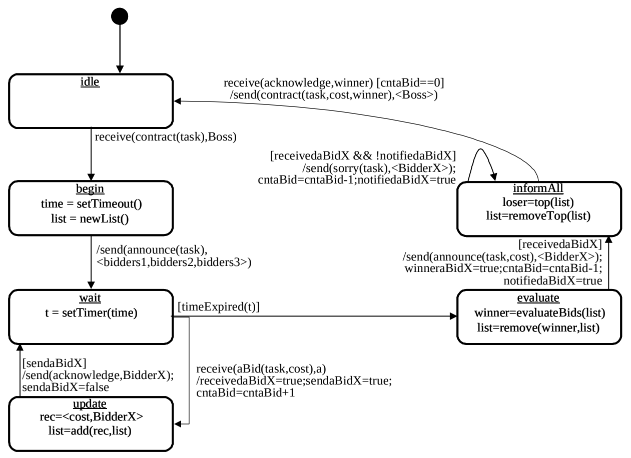
idle
receive(acknowledge,winner) [cntaBid==0]
/send(contract(task,cost,winner),<Boss>)
receive(contract(task),Boss)
[receivedaBidX && !notifiedaBidX]
/send(sorry(task),<BidderX>);
cntaBid=cntaBid-1;notifiedaBidX=true
begin
time = setTimeout()
list = newList()
informAll
loser=top(list)
list=removeTop(list)
/send(announce(task),
<bidders1,bidders2,bidders3>)
[receivedaBidX]
/send(announce(task,cost),<BidderX>);
winneraBidX=true;cntaBid=cntaBid-1;
notifiedaBidX=true
wait
t = setTimer(time)
[timeExpired(t)]
evaluate
winner=evaluateBids(list)
list=remove(winner,list)
[sendaBidX]
/send(acknowledge,BidderX);
sendaBidX=false
receive(aBid(task,cost),a)
/receivedaBidX=true;sendaBidX=true;
cntaBid=cntaBid+1
update
rec=<cost,BidderX>
list=add(rec,list)

Figure 7: Contract Net Concrete Manager Task

Creating a Concrete Behavioral Model

Before a formal model can be automatically generated with Promela, the *abstract* behavior model must be transformed into a *concrete* behavioral model. This transformation will require some user intervention. We explain how this transformation occurs using the Manager and Bidder abstract behavior models from Figures 4 and 5. Figures 7 and 8 contain the concrete behavior models of the Manager and Bidder tasks respectively.

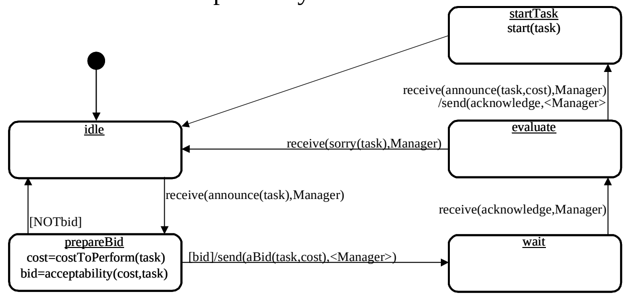
startTask
start(task)
receive(announce(task,cost),Manager)
/send(acknowledge,<Manager>)
idle
receive(sorry(task),Manager)
evaluate
receive(announce(task),Manager)
receive(acknowledge,Manager)
[NOTbid]
prepareBid
cost=costToPerform(task)
bid=acceptability(cost,task)
[bid]/send(aBid(task,cost),<Manager>)
wait

Figure 8: Contract Net Concrete Bidder Task

After the system designer creates the abstract behavior model, the designer invokes agentTool to automatically verify the abstract behavior model. The first step in the verification process is to create a concrete behavior model from the abstract behavior model. The transformation process begins by copying all the states from the abstract behavior model onto a new task design window for the concrete behavior model. The transformation next analyzes each state transition. Every transition from the abstract behavior model will be recreated in the concrete behavior model, but with more detailed information. We are concerned with transforming only *send* actions, *receive* actions, and *guard conditions* as these are the only elements possible on any given transition.

First we look at *send* actions. A *send* action may send a message to one or more recipients. Therefore, we format any send message so the recipient field can be a set of recipients. For example, the send action */send(announce(task),<bidders>)* mandates the *announce* message be sent to one or more *bidders*. When agentTool encounters this type of notation during the abstract to concrete transformation, it prompts the designer for the number of recipients to use during the verification process. After the designer enters a quantity greater than or equal to one, the send action is redefined with a set of recipients as */send(announce(task),<bidders1,bidders2,bidders3>).* From this notation, Promela code can be created automatically.

The next element of a transition we transform is the *receive* action. If the inter-task communication involves a *multicast* message, we define extra variables to keep track of the received messages. We want to know if a particular message is received so we define a boolean variable for each potential received message that matches the sender of that message. For a received message of *aBid* from bidders1 we define a boolean variable *receivedaBid1*. We define similar variables for bidders2 and bidders3. Anticipating we will want to send an *acknowledge* message to the sender of a received message, we define a variable that can be tested from another state. We define the variable the same way we defined the boolean variable for receiving a message. We name the variable *sendaBid1* for receiving the message *aBid* from bidders1. We also define a boolean variable for determining if we have notified a Bidder of winning or losing. This variable also links the message with the Bidder sending an *aBid* message and follows the naming scheme *notifiedaBid1* for an *aBid* message received from Bidder1. Finally, anticipating we might not actually receive *aBid* messages from all the Bidders, we count the number of *aBid* messages received. We define a variable *cntaBid* to keep track of the received *aBid* messages.

The final element of a transition that must be transformed is the guard condition. We analyze each of the guard conditions defined in the abstract behavior model and combine them with the variables we created for multicast communications to produce a concrete behavior model that can be easily converted into Promela. Through research, we observe certain patterns of behavior that can be expected to occur when dealing with multicast communications. Based on these patterns of behavior, we structure the transitions so a simulation can be performed on the behavior model. We have not yet implemented the automatic transformation based on patterns of behavior, but anticipate full implementation during future work.

Creating a Formal Representation

Before a behavior model can be verified, it must be converted into a formal modeling language. We use Promela to create these models. Translating a concrete behavior model into Promela is straightforward. First, we must define the types of messages used in the tasks' interaction. This is done in Promela using an *mtype* declaration that allows a programmer to declare constants as shown below.

mtype = { announce, aBid, acknowledge, sorry, bid, NOTbid, timeExpired, contract };

Next, we must define the channel over which the messages will be sent. The following statement declares a variable `MgrTobidders1` is of the type *chan*, that it does not have a buffer to hold messages, and that messages of type *mtype* can be sent on it.

```
chan MgrTobidders1 = [0] of {mtype};
```

We default the buffer size to zero forcing the synchronization of message passing. Synchronous message passing is a modeling decision that ensures interactions proceed as intended without extra messages being transmitted. Increasing the buffer size creates a FIFO channel that enables us to model asynchronous message passing. All messages have to be taken off the channel in the order they are placed on the channel. If an erroneous message is placed on the channel ahead of a valid message, the valid message will never be read and the interaction deadlocked. The number of interactions defined in the behavior model determines the number of channel declarations.

The next step is to define processes to emulate each side of the interaction. Promela has a construct called a *proctype* that models the task structure for each agent. Each process will contain the states of one task. The idea is to begin the process in the *start* state and end in an acceptable *end* state, while moving from states only if explicitly directed to do so. Figure 7 is the Promela declaration for the behavior model of the Manager task in Figure 4. This task is modeled communicating with three instances of the Bidder task… bidders1, bidders2, and bidders3.

```
proctype ContractNetMgr()
{startState:
   do
   :: goto endIdleState
   od;
 endIdleState:
   do
   :: BossToMgr?contract -> goto beginState
   od;
 beginState:
   do
   :: MgrTobidders1!announce;
      MgrTobidders2!announce;
      MgrTobidders3!announce; goto waitState
   od;
 waitState:
   do
   :: MgrTobidders1?aBid ->
      receivedaBid1 = true; sendaBid1=true;
      cntaBid = cntaBid + 1;
      goto updateState
   :: MgrTobidders2?aBid ->
      receivedaBid2 = true; sendaBid2=true;
      cntaBid = cntaBid + 1;
      goto updateState
   :: MgrTobidders3?aBid ->
      receivedaBid3 = true; sendaBid3=true;
      cntaBid = cntaBid + 1;
      goto updateState
   :: timeExpired -> goto evaluateState
   od;
 updateState:
   do
   :: sendaBid1 -> MgrTobidders1!acknowledge;
      sendaBid1=false; goto waitState
   :: sendaBid2 -> MgrTobidders2!acknowledge;
      sendaBid2=false; goto waitState
   :: sendaBid3 -> MgrTobidders3!acknowledge;
      sendaBid3=false; goto waitState
   od;
 evaluateState:
   do
   :: receivedaBid1 -> MgrTobidders1!announce;
      winneraBid1 = true; notifiedaBid1 = true;
      cntaBid = cntaBid - 1;
      goto informAllState
   :: receivedaBid2 -> MgrTobidders2!announce;
      winneraBid2 = true; notifiedaBid2 = true;
      cntaBid = cntaBid - 1;
      goto informAllState
   :: receivedaBid3 -> MgrTobidders3!announce;
      winneraBid3 = true; notifiedaBid3 = true;
      cntaBid = cntaBid - 1;
      goto informAllState
   od;
 informAllState:
   do
   :: receivedaBid1&&notifiedaBid1 == false ->
      notifiedaBid1=true; MgrTobidders1!sorry;
      cntaBid = cntaBid - 1;
      goto informAllState
   :: receivedaBid2&&notifiedaBid2 == false ->
      notifiedaBid2=true; MgrTobidders2!sorry;
      cntaBid = cntaBid - 1;
      goto informAllState
   :: receivedaBid3&&notifiedaBid3 == false ->
      notifiedaBid3=true; MgrTobidders3!sorry;
      cntaBid = cntaBid - 1;
      goto informAllState
   :: cntaBid==0 && MgrTobidders1?acknowledge;
      BossToMgr!contract; goto endIdleState
   :: cntaBid==0 && MgrTobidders2?acknowledge;
      BossToMgr!contract; goto endIdleState
   :: cntaBid==0 && MgrTobidders3?acknowledge;
      BossToMgr!contract; goto endIdleState
   od;}
```

Figure 7: Promela Code for Manager Task

For an example of how a Promela model works, refer to Figure 7. In the *ContractNetMgr* task is a *wait* state. This state has two possible exit transitions. If an *aBid* message is received via any of the three *MgrTobiddersX* channels, control is transitioned to the *update* state, an *acknowledge* message is sent back out the same channel, and the transition cycles back to the *waitState*. If the guard condition *timeExpired* becomes enabled while control is in the *wait* state, then the task transitions to the *evaluateState*.

Figure 8 is the Promela declaration for one of the Bidder tasks in Figure 5 that is communicating with the Manager task in Figure 4.

```
proctype ContractNetBidder1()
{startState:
   do
   :: goto endIdleState
   od;
 endIdleState:
   do
   :: MgrTobidders1?announce ->
      goto prepareBidState
   od;
 prepareBidState:
   do
   :: bid -> MgrTobidders1!aBid; goto waitState
   :: NOTbid -> goto endIdleState
   od;
 waitState:
   do
   :: MgrTobidders1?acknowledge; goto
      evaluateState
   od;
 evaluateState:
   do
   :: MgrTobidders1?announce ->
      MgrTobidders1!acknowledge;
      goto startTaskState
   :: MgrTobidders1?sorry; goto endIdleState
   od;
 startTaskState:
   do
   :: goto endIdleState
   od;}
```

Figure 8: Promela Code for Bidder Task

The keyword *proctype* declares a process. The States begin with a label followed by a colon. The *do..od* loops trap the flow of control inside their respective states. You can only exit a *do..od* loop with a *goto* statement or a *break* statement. The *goto* transfers control to another state while the *break* just exits the loop and falls through into the next state. For obvious reasons, it is unacceptable to fall into another state unless explicitly directed to do so. An exclamation point (!) after the channel variable *MgrToAgent1* signifies the *announce* message has been placed on the channel. A question mark (?) after the channel variable *MgrToAgent1* signifies the message following the question mark is taken off the channel via a receive action if it has been placed on the channel. The arrow (->) is a statement separator and serves as an implication symbol. If the statement before the arrow is executed then the statement after the arrow is also executed. The semicolon (;) is also a statement separator but carries no implications.

Once all the tasks' behavior models have been created, we define a process to start the conversation processes called an *init* process. The keyword *atomic* mandates all statements enclosed within its brackets will be executed without interruption by external processes. The keyword *run* starts the processes running in parallel. Figure 9 shows the *init* procedure for starting five tasks, a Boss, Manager, and three Bidders.

Verification

We can now use Spin to check for interaction errors. The type of error we detect is deadlock. Spin will create an analyzer to search the entire state space of the tasks' interaction, simulating every possible combination of messages in the interaction until either a deadlock condition occurs or the state space is exhausted. Task interactions are considered deadlocked if they stop executing in any state other than a valid *end* state (marked by the task designer). If a deadlock condition is detected, the analyzer writes a trace file that can be used to create a message sequence trace pinpointing the series of message events that led to the deadlock.

```
init
{atomic
   { run ContractNetBoss();
      run ContractNetMgr();
      run ContractNetBidder1();
      run ContractNetBidder2();
      run ContractNetBidder3();}}
```

Figure 9: Init Procedure for ContractNet Protocol

Error Detected

The *ContractNet* protocol as modeled in Figures 4 and 5 contains an error. If the Manager task sends multiple *announce* messages while transitioning from the *start* state to the *wait* state, and the *timeExpired* guard condition in the *wait* state becomes enabled before all of the Bidders have had a chance to respond, then the Bidder tasks that want to place a bid cannot. This is because the Manager task is not in a state that will accept more bids. Additionally, Bidder agents that tried to respond with a late bid and transmitted a bid to the Manager task have now hung themselves up because the transmitted message cannot be received. This error condition was detected using our methodology with Promela and Spin and is shown in Figure 10.

```
proc 0 = :init:
proc 1 = ContractNetBoss
proc 2 = ContractNetMgr
proc 3 = ContractNetBidder1
proc 4 = ContractNetBidder2
proc 5 = ContractNetBidder3
q\p   0   1   2   3   4   5
  1   .   BossToMgr!contract
  1   .   .   BossToMgr?contract
  2   .   .   MgrTobidders1!announce
  2   .   .   .   MgrTobidders1?announce
  3   .   .   MgrTobidders2!announce
  3   .   .   .   .   MgrTobidders2?announce
  4   .   .   MgrTobidders3!announce
  4   .   .   .   .   .   MgrTobidders3?announce
  4   .   .   .   .   .   MgrTobidders3!aBid
  4   .   .   MgrTobidders3?aBid
  4   .   .   MgrTobidders3!acknowledge
  4   .   .   .   .   .   MgrTobidders3?acknowledge
  3   .   .   .   .   MgrTobidders2!aBid
  3   .   .   MgrTobidders2?aBid
  3   .   .   MgrTobidders2!acknowledge
  3   .   .   .   .   MgrTobidders2?acknowledge
  3   .   .   MgrTobidders2!announce
  3   .   .   .   .   MgrTobidders2?announce
  4   .   .   MgrTobidders3!sorry
  4   .   .   .   .   .   MgrTobidders3?sorry
  3   .   .   .   .   MgrTobidders2!acknowledge
  3   .   .   MgrTobidders2?acknowledge
  1   .   .   BossToMgr!contract
  1   .   BossToMgr?contract
spin: trail ends after 58 steps
-------------
final state:
-------------
#processes: 6
                 receivedaBid1 = 0
                 receivedaBid2 = 1
                 receivedaBid3 = 1
                 winneraBid1 = 0
                 winneraBid2 = 1
                 winneraBid3 = 0
                 sendaBid1 = 0
                 sendaBid2 = 0
                 sendaBid3 = 0
                 notifiedaBid1 = 0
                 notifiedaBid2 = 1
                 notifiedaBid3 = 1
                 cntaBid = 0
 58:     proc  5 (ContractNetBidder3) line 153 "Goverify"
(state 7) <valid endstate>
 58:     proc  4 (ContractNetBidder2) line 122 "Goverify"
(state 7) <valid endstate>
 58:     proc  3 (ContractNetBidder1) line  96 "Goverify"
(state 11)
 58:     proc  2 (ContractNetMgr) line  42 "Goverify"
(state 7) <valid endstate>
 58:     proc  1 (ContractNetBoss) line  33 "Goverify"
(state 19) <valid endstate>
 58:     proc  0 (:init:) line 187 "Goverify" (state 7)
<valid endstate>
6 processes created
```

Figure 10: Error Trace for Contract Net Protocol

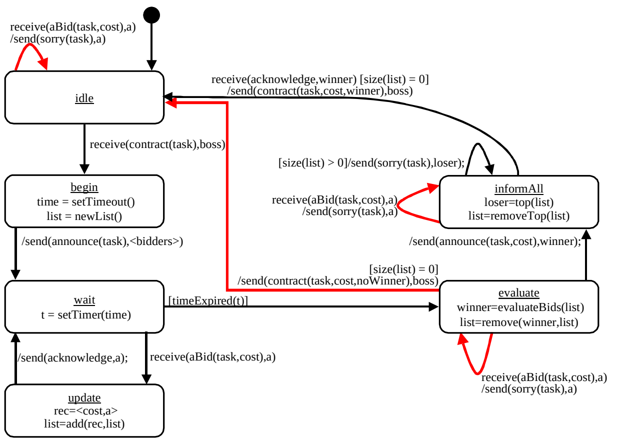
receive(aBid(task,cost),a)
/send(sorry(task),a)
idle
receive(acknowledge,winner) [size(list) = 0]
/send(contract(task,cost,winner),boss)
receive(contract(task),boss)
[size(list) > 0]/send(sorry(task),loser);
receive(aBid(task,cost),a)
/send(sorry(task),a)
begin
time = setTimeout()
list = newList()
informAll
loser=top(list)
list=removeTop(list)
/send(announce(task),<bidders>)
/send(announce(task,cost),winner);
[size(list) = 0]
/send(contract(task,cost,noWinner),boss)
wait
t = setTimer(time)
[timeExpired(t)]
evaluate
winner=evaluateBids(list)
list=remove(winner,list)
/send(acknowledge,a);
receive(aBid(task,cost),a)
receive(aBid(task,cost),a)
/send(sorry(task),a)
update
rec=<cost,a>
list=add(rec,list)

Figure 11: Corrected ContractNet Manager Task

Error Corrected

Correcting the *ContractNet* protocol required changes in both the Manager and Bidders tasks. Since it is possible for Bidders to send *aBid* messages to the Manager after the Manager has finished waiting for messages, we must be prepared to handle late messages. The Manager task was changed to send *sorry* replies back to any Bidder placing a bid after the timeout had occurred. It is also possible that no Bidders actually place a bid. Therefore, another fix was to place a transition from the *evaluate* state back to the *idle* state in the case no bids were received before the time expired. Figure 11 shows the corrected task diagram for the Manager task.

The Bidder task was changed to receive a *sorry* message in the case a late bid was sent to the Manager task. This allows the Bidder task to return to its normal *idle* state. Figure 12 shows the corrected Bidder task diagram.

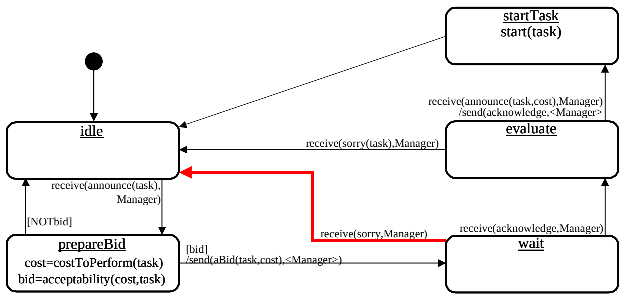
startTask
start(task)
receive(announce(task,cost),Manager)
/send(acknowledge,<Manager>
idle
evaluate
receive(sorry(task),Manager)
receive(announce(task),
Manager)
[NOTbid]
receive(sorry,Manager)
receive(acknowledge,Manager)
prepareBid
cost=costToPerform(task)
bid=acceptability(cost,task)
[bid]
/send(aBid(task,cost),<Manager>)
wait

Figure 12: Corrected ContractNet Bidder Task

Feedback

All errors detected by agentTool are displayed graphically by highlighting the state and/or transition that caused the error. When an error condition occurs, Spin generates a trace file that can be used to recreate the simulation that detected the error. Using this simulation, we can pinpoint the states and often the exact transitions that are causing the problems. This feature has not been fully implemented for tasks in agentTool, but has been implemented for verifying conversations (Lacey, 2000).

Conclusions

The automatic verification of agent tasks' interactions is possible with our methodology. Ongoing research will determine the patterns of behavior required to predict the correct modeling of complicated agent interaction protocols such as the Contract Net protocol and various auction protocols. These patterns of behavior will be used to make the transformation from an abstract behavioral model to a concrete behavioral model as seamless as possible, with a minimum of user input. Finally, the automatic generation and analysis of Promela code from state transition diagrams has been demonstrated by (Lacey, 2000) and a similar process is used here to verify behavioral models.

Related Research

Spin is a generic verification system and has been used extensively to verify real-life problems such as algorithms, communications network design problems, and protocol design problems (Holzmann, 1997). Some agent researchers are looking at how to represent conversations with formal languages and how to verify a model meets a specification (Greaves, 1999). FIPA has taken measures to publish "verifiably correct" protocols that, if implemented correctly, will work as published (FIPA, 1998). Some groups define agent conversations with finite state machines, convert them manually to formal languages, and then mathmatically prove them correct (Martin, 1999). However, this is the only research we know that allows a system designer to graphically design agent interactions, automatically verify properties of multiagent systems such as agent conversations and agent task interaction, and automatically provide feedback to the system designer pinpointing the source of error conditions.

Summary

This paper describes the methodology used to automatically verify agent behavioral models in a multiagent system. The process begins by modeling the agent interactions as tasks with state transition diagrams in agentTool using the MaSE methodology. Abstract behavioral models are then semi-automatically translated into concrete behavioral models which are then converted into Promela code that is analyzed by Spin for deadlock errors. Feedback on errors is provided to agentTool users through text messages and graphical highlighting.

Acknowledgements

The Air Force Office of Scientific Research supported this research. We thank Captain Alex Kilpatrick for his financial support and latitude provided in this endeavor. We also thank Tom Hartrum for his support and advice.

References

DeLoach, Scott A. Multiagent Systems Engineering: A Methodology and Language for Designing Agent Systems, *Proceedings of a Workshop on Agent-Oriented Information Systems (AOIS '99).* 45-57. Seattle, WA. May 1, 1999.

Foundation for Intelligent Physical Agents. Agent Communication Language. FIPA 97 Specification, Version 2.0. October 1998.

Greaves, M. and others. *Agent Conversation Policies, Handbook of Agent Technology*. Cambridge: AAAI Press/ MIT Press, 1999.

Holzmann, Gerard J. The Model Checker Spin, *IEEE Transactions On Software Engineering, Volume 23, Number 5*: 279-295 (May 1997).

Lacey, Timothy H. and Scott A. DeLoach. "Automatic Verification of Multiagent Conversations," Submitted to Proceedings of the Midwest Artificial Intelligence and Cognitive Science Conference. 2000.

Martin, Francisco J., Enric Plaza, and Juan A. Rodriguez-Aguilar. "Conversation Protocols: Modeling and Implementing Conversations in Agent-Based Systems," Proceedings of the Automomous Agents '99 Workshop on Specifying and Implementing Conversation Policies. 49-58. Seattle, 1999.

Pressman, Roger S. *Software Engineering: A Practitioner's Approach.* New York: McGraw-Hill, 1997.

Sycara, Katia P. Multiagent Systems, *AI Magazine*: 79-92, (Summer 1998).