Automatic Verification of Multiagent Conversations[◆]

Timothy 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
timothy.lacey@afit.af.mil
scott.deloach@afit.af.mil

Abstract

As network bandwidth increases, distributed applications are becoming increasingly prevalent. Systems using these applications are very complicated to build and must be dependable. Software agents are ideal for breaking complicated problems into manageable subtasks. Agent conversations, a series of messages passed between agents, are the cornerstone of multiagent systems and must be deemed correct before being placed into service. This paper introduces a method to automatically verify that conversations are valid before employing them. Agent conversations are created graphically using state transition diagrams in the agentTool multiagent development environment. This graphical representation is then transformed into a formal modeling language called Promela that is analyzed by the Spin verification tool to detect errors such as deadlock, non-progress loops, syntax errors, unused messages, and unused states. Feedback is provided to the user automatically via text messages and graphical highlighting of error conditions.

Introduction

As network bandwidth increases, distributed applications are becoming increasingly prevalent. The systems required to run these applications are very complicated to build. Companies have huge investments in their systems, and depend upon them greatly. Therefore, it's understandable these systems must be robust and verifiably correct.

Intelligent software agents are also becoming more popular. Distributed agents can be used to retrieve, filter, and summarize information as well as provide intelligent user interfaces – just to name a few of the many applications suited to software agents. Because of their distributed nature, intelligent software agents are an appropriate mechanism for solving complicated problems.

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. However, not all multiagent systems are open systems. Closed agent systems are those where agents interact with each other via structured and predictable 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 methods between the agents must be formally verified. For example, errors in conversations can prevent orders from getting through to subordinates or financial transactions from being completed. The verification process includes checking for infinite loops, deadlocks, and other communication pitfalls that would prevent a multiagent system from completing its mission. This paper introduces a formal methodology that automatically verifies that a system of agents will communicate as expected before a user deploys the system. Then, and only then, can the user of the multiagent system have assurance the system will communicate 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

Agents communicate with each other using patterns of messages called *conversations* (Greaves, 1999). In our methodology, 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

[◆] 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.

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).

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

During the domain level design, the communications between agents are specified as conversations using state transition diagrams. The system uses an automated verification tool (Spin) and formal modeling language (Promela) to verify these conversations are valid. Feedback is provided to the user indicating whether the conversation design is valid.

Promela/Spin

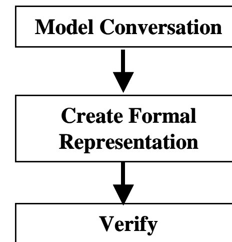
Agent conversations are specified in Promela (Holzmann, 1997). Promela differs from other formal languages in that it is a modeling language. As such, it is used to abstractly model communication protocols. Conversations are modeled as processes, conversation paths are modeled as channels, and variables can be visible to the entire conversation (global) or visible only to a particular portion of a conversation (local). All statements are either *executable* or *blocked*. Statements are blocked if the statement is a conditional statement and the condition is false. In this case, the statement blocks until the condition becomes true. Statements are also blocked when waiting to receive messages. This property provides a means of synchronizing communications between processes by causing one process (a responder) to wait on a message sent by another process (the initiator) while in a specific state. The initiating process may also block while waiting on a reply from the responding process.

Spin is an automated verification tool from Bell Labs that operates on the Promela modeling language. Spin will detect deadlock, livelock, assertion violations, and many other communication centric errors while efficiently using computer resources.

Automatic Verification of Conversations

The present method of protocol verification requires a human to manually model a protocol in a formal language so the verifier can be used. Most people believe formal methods are too difficult to understand and use in this manner (Hinchey, 1999). The challenge then is to automatically generate the formal representation of a conversation and then use an automated tool to verify that this representation is free from undesirable communication properties such as deadlock, livelock, and infinite loops. Figure 1 is a top-level view of the overall process.

Figure 1: Top Level View of Methodology



Modeling Agent Conversations

In agentTool, we assume that all agent conversations are binary and consist of an *initiator* side and a *responder* side. Both sides of the conversation move through various states in harmony as the conversation develops. Eventually, both sides of the conversation should end up in their respective "end" states and the conversation is complete. The state transition diagram allows us to visualize the various states that a conversation goes through and the events that cause the conversation to move from state to state.

Figure 2 illustrates one side of a typical conversation while Figure 3 illustrates the complimentary side. The two sides make up one complete conversation, which may be part of a much larger system (or set) of conversations.

The beginning state in a conversation is the "start" state, signified by a solid circle. The final state in a conversation is the "end" state and is signified by a solid circle with a ring drawn around it. Each intermediary 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 Unified Modeling Language (UML) notation (Rational, 1997) event-name (argument list) [guard condition] /action-expression ^send-clause.

The label may contain some or all of this information. Each state may have more than one entry point and exit point, but all exit points must be deterministic in that at any given point in time only one exit point is enabled and can be used. If more than one exit point is enabled at any given time, unpredictability is the result and the conversation is no longer valid.

In Figure 2 there are three states: *start*, *wait* and *end*. The transition from the *start* state to the *wait* state sends a *send(information)* message. The information is a

parameter that is passed with the message. The transition from the *wait* state back to itself takes place when a *failure-transmission* message is received while in the *wait* state. This transition receives a *failure-transmission* message then sends a *send(information)* message before transitioning back to the *wait* state. Finally, the transition from the *wait* state to the *end* state takes place when an *acknowledge* message is received while in the *wait* state.

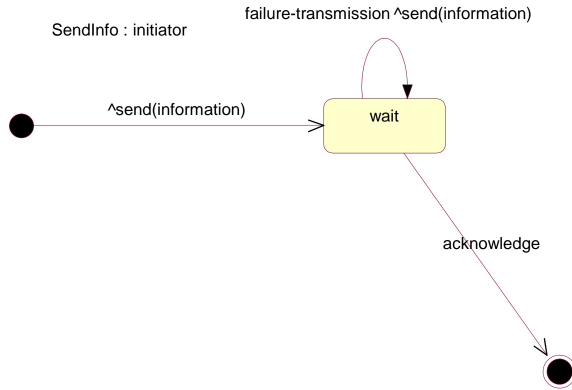


Figure 2: Initiator Half of Conversation SendInfo (DeLoach, 1999)

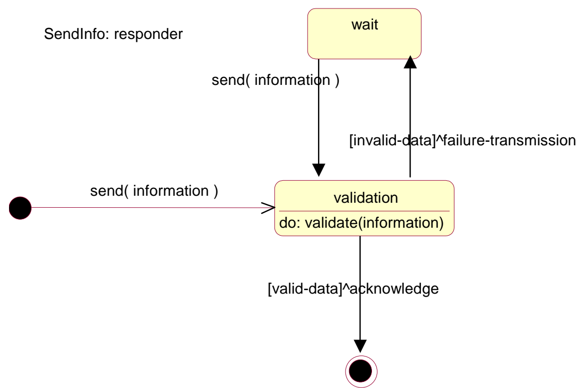


Figure 3: Responder Half of Conversation SendInfo (DeLoach, 1997)

The responder side of the *SendInfo* conversation (Figure 3) has four states and the transitions complement the transitions in the initiator side of the conversation. The next step in the modeling process is to convert the above state transition diagrams into its formal representation in Promela.

Creating a Formal Representation

The first step in translating the state transition diagram to Promela is to create an intermediary state table. A state table is a textual representation of a graphical state transition diagram and has the advantage that it can be parsed easily. Although not mandatory, it simplifies the job of creating a formal representation.

The state table is built from the transition labels on the transition arrows of a state transition diagram. The state table is simply an ordering of all possible transitions in a state transition diagram. The format of the state table mirrors that of the transition labels in a state transition diagram. However, each entry in the state table must also specify the state the transition is coming from and the state to which it is going. The different fields of a state table entry are separated by a semicolon for ease in parsing. Figure 4 illustrates a state table using the *SendInfo* conversation defined in Figures 2 and 3. In this state table, a name is given to both halves of the conversation and this name inserted at the beginning of each line. Each entry in the state table contains a *process name* (consisting of the *conversation name* and the *participant's name*), *current state*, *received message*, *guard condition*, *transmitted message*, and *next state*, and must be unique.

```
SendInfoResponder;startState;send;
  null;null;validationState
SendInfoResponder;validationState;
  null;invalidData;
  failureTransmission;waitState
SendInfoResponder;validationState;
  null;validData;acknowledge; endState
SendInfoResponder;waitState;send;
  null;null;validationState
SendInfoResponder;endState;null; null;null;null
SendInfoInitiator;startState;null;
  null;send;waitState
SendInfoInitiator;waitState;
  failureTransmission;null;send; waitState
SendInfoInitiator;waitState;
  acknowledge;null;null;endState
SendInfoInitiator;endState;null; null;null;null
```

Figure 4: State Table of Conversation SendInfo

Modeling a conversation in Promela is straightforward. After parsing the state table, the Promela code can be automatically generated. First, we must define the types of messages used in the conversation. This is done in Promela using an *mtype* declaration that allows a programmer to declare constants as shown below.

```
mtype={send,acknowledge};
```

These values are found by searching through the state table and creating a vector of messages by examining the *received message*, *guard condition*, and *transmit message* fields.

Next, we must define the channel over which the messages will be sent. The statement

```
chan bus1 = [0] of {mtype};
```

states that a variable *bus1* 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. All messages have to be taken off the channel (received) before another message can be placed on the channel (transmitted). Synchronous message passing is a modeling decision that ensures conversations proceed as intended without extra messages being transmitted. The number of channel declarations is determined by the number of conversations defined in the

state table. If only one conversation is in the state table (as in this example), then only one channel declaration must be made.

The next step is to define processes to emulate each side of the conversation. Promela has a construct called a *proctype* that models each half of a conversation. Each process will contain the states of one half of the conversation. The idea is to begin the process in the *startState* and end in the *endState*, while moving from states only if explicitly directed to do so. Figure 5 shows the *proctype* declaration for the *responder* side of the *SendInfo* conversation, while Figure 6 shows the *initiator* side of the same conversation.

```
proctype SendInfoResponder()
{
  startState:
    do
    :: bus1?send -> goto validationState
    od;
  validationState:
    do
    :: invalidData ->
     bus!failureTransmission; goto waitState
    :: validData -> bus1!acknowledge; goto
   endState
    od;
  waitState:
    do
    :: bus1?send -> goto validationState
    od;
  endState:
    do
    :: break
    od;
}
```

Figure 5: Process SendInfoResponder

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 *bus1* signifies the message *send* has been placed on the channel. A question mark (?) after the channel variable *bus1* 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.

```
proctype SendInfoInitiator()
{
  startState:
    do
    :: bus1!send -> goto waitState
    od;
  waitState:
    do
    :: bus1?failureTransmission -> bus!send; goto
   waitState
    :: bus1?acknowledge -> goto endState
    od;
  endState:
    do
    :: break
    od;
}
```

Figure 6: Process SendInfoInitiator

For example, looking at the *validationState* in Figure 5, once the conversation has entered this state it will stay there via the *do..od* loop. While in this state, if the guard condition *invalidData* becomes true then a *failureTransmission* message is sent and the conversation goes to the *waitState*.

Conversely, if the guard condition *validData* becomes true, an *acknowledge* message is sent and the conversation goes to the *endState* where the conversation terminates. Since the guard conditions are declared as *mtype* variables, Spin treats them as messages that are not transmitted on a channel. If more than one guard condition exists in a given state, Spin will arbitrarily choose one of the statements to execute and block the others simulating the changing of guard conditions.

Once the two halves of the conversation have been modeled, we create a process to start the conversation processes called an *init* process. Figure 7 shows the *init* process for the *SendInfo* conversation.

```
init
{  atomic
   {  run SendInfoResponder();
      run SendInfoInitiator() }
}
```

Figure 7: Init Process for SendInfo Conversation

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.

Verifying Message Sequences. Sequence diagrams (Rational, 1997) are beneficial for real-time specifications and for complex scenarios. They show the explicit sequence of messages between agents and can exist in a generic form (all the possible sequences of messages) or an instance form (one actual sequence consistent with the generic form). Sequence diagrams show us the big picture in the grand scheme of agent conversations.

A message sequence is created by listing desired messages between conversations in a specified order.

Sequence diagrams represent interactions among agents within a system to achieve a desired operation or result. A graphical representation of a message sequence is called a message sequence chart (Rational, 1997). Figure 8 shows a valid message sequence chart encompassing two conversations (SendInfo and CollectData) between three agents (Commander, Mission Cntrl, and Data Collection). Not all of the messages that could be sent in these conversations need be included in the message sequence chart.

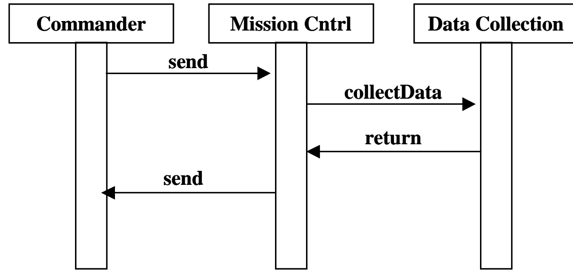


Figure 8: Message Sequence Chart

Message sequences are converted to a table similar to a state table as shown in Figure 9. The format of the message sequence table is *Conversation Name*; *Conversation From Participant*; *Conversation To Participant*; *Message*. When checking for a message sequence the sequence is defined in a Promela *never* claim and checked for its existence. A *never* claim is a special type of process that is optional and if exists is used to detect undesirable behavior. If a message sequence defined in a *never* claim is found, Spin will generate an error. Of course, this is not really an error because we want to verify the sequence exists and the error condition has confirmed the sequence does indeed exist. Figure 10 is the *never* claim for the message sequence table of Figure 9.

```
SendInfo;Responder;Initiator;send
CollectData;Initiator;Responder;collectData
CollectData;Responder;Initiator;return
SendInfo;Initiator;Responder;send
```

Figure 9: Message Sequence Table

A key difference in the modeling of a message sequence and a conversation is the way message events are detected. In a conversation, the channel that messages are transmitted on is constantly monitored and messages must be placed on the channel and taken off the channel in a predetermined order. In a message sequence, the channel is monitored but only messages that we are looking for are detected.

Many messages may be placed on the channel and taken off the channel before the desired message is detected as part of a particular sequence. Modeling sequences in this fashion provides great flexibility in detecting message sequences that span multiple conversations.

```
never
{
   State0:
      do
      :: SendInfo?[send] -> goto State1
      :: skip
      od;
   State1:
      do
      :: CollectData?[collectData] -> goto State2
      :: skip
      od;
   State2:
      do
      :: CollectData?[return] -> goto State3
      :: skip
      od;
   State3:
      do
      :: SendRawIntel?[send] -> goto State4
      :: skip
      od;
   State4:
      do
      :: SendInfo?[send] -> goto accept
      :: skip
      od;
   accept:
      skip
}
```

Figure 10: Message Sequence Verification

After generating the Promela source code, it is used as input to the verification tool Spin as discussed in the next section.

Verification

We can now use Spin to check for conversation errors. The first type of error we detect is deadlock. Spin will create an analyzer to search the entire state space of the conversation, simulating every possible combination of messages in the conversation until either a deadlock condition occurs or the state space is exhausted. Conversations are considered deadlocked if they terminate in any state other than the *end* state. 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.

The next type of error detected is non-progress loops. We mark all states in the conversation with the keyword progress, which Spin uses to check that all states are entered during at least one execution path. If a state is not entered into then a non-progress error is generated. Many things can cause a non-progress error to include livelock, infinite overtaking, deadlock, unused states and unused transitions.

Finally we test for valid message sequences by defining the message sequence in a *never* claim and checking to see if the sequence exists. If the sequence exists, Spin generates a *never* claim violation error. However, this is not really an error since we want the sequence to exist. If Spin does not report a *never* claim violation, the message

sequence could not be found and even though the conversations as defined may be valid, the required message sequence is not contained therein.

The *SendInfo* conversation does not contain any errors. However, to demonstrate the types of error messages Spin would generate, we create a deadlock condition by changing the transmitted message *acknowledge* from the *validation* state in Figure 3 to a received message. Figure 11 shows the error messages generated after analyzing the flawed conversation.

```
pan: invalid endstate (at depth 5)
pan: wrote verify.trail
(Spin Version 3.2.4 -- 10 January 1999)
Warning: Search not completed
   + Partial Order Reduction
Full statespace search for:
   never-claim          - (none specified)
   assertion violations +
   cycle checks         - (disabled by -DSAFETY)
   invalid endstates +
State-vector 24 byte, depth reached 8, errors: 1
       6 states, stored
       1 states, matched
       7 transitions (= stored+matched)
       1 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)
1.493    memory usage (Mbyte)
```

Figure 11: Spin Output of Flawed SendInfo Conversation

The first line of the error message tells us the conversation ended in an *invalid endstate*, meaning the conversation terminated in one of the states other than the *end* state. The other messages provide extraneous information that does not help us actually find the error. However, the second line of the messages tells us a file, *verify.trail* was written. This file can be analyzed by Spin and a message sequence trace created pinpointing the exact location of the deadlock condition. Figure 12 portrays the messages agentTool provides the system user.

```
DEADLOCK CONDITION EXISTS IN THE FOLLOWING
   CONVERSATION:
        Conversation Name = SendInfo
        Participant Name = Responder
        Current State = validation
        State Transition = null
DEADLOCK CONDITION EXISTS IN THE FOLLOWING
   CONVERSATION:
        Conversation Name = SendInfo
        Participant Name = Initiator
        Current State = wait
        State Transition = failureTransmission
```

Figure 12: agentTool Error Messages

All errors detected by agentTool are displayed graphically by highlighting the state and/or transition that caused the error. Figure 13 is the responder side of the *SendInfo* conversation with the *end* state highlighted. Though not shown here, the initiator side of the conversation's *end* state is also highlighted. The *end* states are highlighted because they were never entered into. The *failureTransmission* transition on the initiator side of the *SendInfo* conversation is also highlighted as indicated in Figure 12 because it is the source of the deadlock.

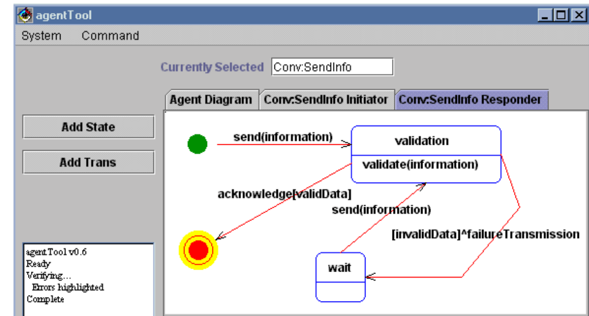


Figure 13: Error Highlighting in SendInfo Conversation

Analysis of Types of Errors Handled. Spin can check for many types of errors (Holzmann, 1997). Unfortunately, agentTool does not currently provide the capability to enter the required data to check for all of them. However, agentTool does allow the checking of a few classic conversation centric errors.

Detectable Errors. *Deadlocks* are detected with complete certainty. This is accomplished by performing an exhaustive state space search for deadlock conditions. If a deadlock exists in a single conversation, agentTool and Spin will detect it.

Non-progress loops are detected. This can also be called *livelock* or *infinite overtaking* because the error condition that results is the same for all three cases. The error condition that results from any of these three conditions is that at least one state in the conversation is not entered. Therefore, that state is labeled a *non-progress* state and an error is generated.

Unused states are detected by checking for non-progress loops. If a state is not used, it is not entered into and a *non-progress* error is generated.

Unused messages are detected when they are not taken off the message channel, thereby leaving messages on the buffer. Since messages placed on the channel must be matched by a receiving process that takes them off the buffer, any unused messages will generate deadlock errors. This might not be a deadlock condition, but the error raised will generate enough information for the user to identify the source of the problem.

Mislabeled transitions are detected when Spin is first run. If the syntax is incorrect, Spin cannot compile the Promela code into the executable analyzer. Feedback is provided via a message window when a syntax error occurs. Mislabeled transitions can also be syntactically correct but create deadlocks or *non-progress* loops. Though the indication is not of a syntactical error, the error

is still caught and enough information provided to the user to determine the cause of the error.

Inability to create required sequences is detected using never claims. The desired message sequence is modeled using a *never* claim, and if Spin does not generate a *never* claim violation, the message sequence does not exist.

Undetectable Errors. There are some communication errors that agentTool and Spin cannot currently detect. These errors would be difficult for any automated system to detect; however, they are mentioned here to make the user aware of this limitation.

Timing errors caused by system properties cannot be detected by Spin. The conversations may be valid, but if a system property causes a conversation to pause indefinitely, the complementary conversation is deadlocked until the system property allows the conversation to continue. In this scenario, the conversations are valid and have been verified. Nevertheless, the overall system will not perform correctly.

Hardware failures that cause infinite conversation loops cannot be detected by agentTool and Spin. The conversations are valid and have been verified, but if a sensor or other piece of hardware continues to send the same message in the context of a valid conversation, the conversation can become livelocked and the conversation cannot progress.

Guard conditions specified incorrectly cannot be detected by agentTool and Spin. If a guard condition is specified as part of a conversation, agentTool uses a figurative representation of the guard condition to verify the conversation. If the guard condition consists of an algebraic formula that is incorrectly written, or an incorrectly written logical formula, Spin and agentTool will never know.

Interacting conversations deadlocked when both are contending for a common resource. Even though the conversations are valid, they can deadlock waiting for the same resource.

We plan to implement a *syntax checker* in agentTool that will detect many of these errors such as state transition diagrams and guard conditions that are incorrectly specified.

Agent Communication Languages

An agent communication language (ACL) enables similar software agents to communicate with each other via predefined performatives. A performative specifies the format of any given message and dictates how an agent should respond to messages. Two popular communication languages are the Knowledge Query and Markup Language (KQML) (Bradshaw, 1995) and the Foundation for Intelligent Physical Agents Agent Communication Language (FIPA, 1997).

The choice of an ACL does not impact the automatic verification of conversations. After the conversations in an agent system have been verified, the system may be deployed. Before deploying the system, the desired ACL is chosen and messages between agents are formatted accordingly. For example, if the KQML ACL were chosen, the *SendInfo* conversation would be converted into a set of KQML messages as shown in Figure 14.

```
(send()
   :sender  agent1
   :content ()
   :receiver       agent2)
(failureTransmission()
   :sender  agent2
   :content ()
   :receiver       agent1)
(acknowledge()
   :sender  agent2
   :content ()
   :receiver       agent1)
```

Figure 14: KQML Implementation

Parameters associated with the message are included in the performative field. Data associated with the message is included in the content field. This implementation of KQML is not a *standard* one. However, KQML allows flexibility in its implementation. Since this is a closed agent system, only the participating agents need to know how to interpret the format of the KQML messages.

Summary

This paper describes the methodology used to automatically verify agent conversations in a multiagent system. The process begins by modeling the conversations with state transition diagrams in agentTool using the MaSE methodology. State transition diagrams are then converted into Promela code that is analyzed by Spin for deadlock and *non-progress* errors. We also show how Promela and Spin can be used to verify message sequences by declaring a *never* claim and checking for its existence. Finally, we analyze the types of errors that can and cannot be detected using this methodology. Feedback on errors is provided to agentTool users through text messages and graphical highlighting.

Acknowledgements

This research was supported by the Air Force Office of Scientific 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

Bradshaw, Jeff. *Software Agents*. Cambridge: MIT Press, 1995.

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 (FIPA). *Agent Communication Language*. FIPA 97 Specification, Version 2.0. Geneva, Switzerland, 1997.

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).

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

Rational Software Corporation. *Unified Modeling Language Notation Guide*. Version 1.1, 1 September 1997.

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