Presented at Agent-Oriented Information Systems (AOIS) ’99

Multiagent Systems Engineering: A Methodology And Language for Designing Agent Systems

Scott A. DeLoach
Department of Electrical & Computer Engineering
Air Force Institute of Technology
2950 P Street, Wright-Patterson AFB, OH 45433-7765
Scott.Deloach@afit.af.mil

1. Introduction

Much of the current research related to intelligent agents has focused on the capabilities and structure of individual agents. However, in order to solve complex problems, these agents must work cooperatively with other agents in a heterogeneous environment. This is the domain of *Multiagent Systems*. In multiagent systems, we are interested in the coordinated behavior of a system of individual agents to provide a system-level behavior. Sycara [13] lists six challenges of multiagent systems:

1. How to decompose problems and allocate tasks to individual agents.
2. How to coordinate agent control and communications.
3. How to make multiple agents act in a coherent manner.
4. How to make individual agents reason about other agents and the state of coordination.
5. How to reconcile conflicting goals between coordinating agents.
6. How to engineer practical multiagent systems.

Our research in *Multiagent Systems Engineering* (MaSE) is an attempt to answer the sixth challenge, how to engineer practical multiagent systems, and to provide a framework for solving the first five challenges. It uses the abstraction provided by multiagent systems for developing intelligent, distributed software systems. A second goal of this research is to define a methodology specifically for formal agent system synthesis. To accomplish the first goal, MaSE uses two languages to describe agents and multiagent systems: the Agent Modeling Language (AgML) and the Agent Definition Language (AgDL). AgML is a graphically based language that describes the types of agents in a system and their interfaces to other agents. AgDL is based on first order predicate logic and is used to completely describe the internal behavior of each individual agent. To help achieve the second goal, both AgML and AgDL will be defined with a precise, formal semantics. Although the MaSE methodology has been developed to support formal system synthesis, it does not restrict the use of MaSE to formal development. The methodology can also be successfully applied with traditional software implementation techniques as well.

In this research, we view MaSE as a further abstraction of the object-oriented paradigm where agents are at an even higher level of abstraction than typical objects. Instead of simple objects, with methods that can be invoked by other objects agents coordinate their actions via conversations to accomplish individual and community goals. Interestingly, this viewpoint sidesteps the issues regarding what is or is not an *agent*. We view agents merely as a convenient abstraction, which may or may not possess intelligence. In this way, we handle intelligent and non-intelligent system components equally within the same framework.

This paper overviews MaSE and provides a high-level introduction to one critical component used within MaSE, the Agent Modeling Language. Details on the Agent Definition Language and detailed agent design are left for a future paper.

1.1. Objects to Agents

Agents are typically perceived to have at least four basic traits. Using traditional definitions [14] agents are

- Autonomous – they are not controlled directly by humans or others.
- Cooperative – which implies communication.
- Perceptive – they perceive their environment and react to it. They can also affect their environment.
- Pro-active – they exhibit goal-directed behavior.

While similar to objects, these four traits imply characteristics that objects generally do not have. There are two basic differences [10]:

- Objects are passive. They react to external stimuli, but do not exhibit goal directed behavior.
- Agents typically use a common messaging language between all agents whereas object messages are usually class dependent.

From the four basic characteristics defined above, we see that, despite their differences, we can model agents as "active objects". In other words, we view agents as an object with goals and a common communication language. Therefore, to develop our MaSE methodology, we build upon existing object oriented analysis and design techniques such as Rumbaugh's Object Modeling Technique (OMT) [9] and the Unified Modeling Language (UML) [8]. While MaSE diagrams may look similar to OMT or UML diagrams, we have added additional features and modify traditional object-oriented semantics to capture notions of agency and cooperative behavior. Also, the formal definitions of AgML and AgDL make system synthesis more straightforward than when using informally specified object oriented definition languages and methodologies such as OMT or UML [2].

2. MaSE Methodology

The MaSE methodology is similar to traditional software engineering methodologies is but specialized for use in the distributed agent paradigm. The methodology follows the basic steps shown in Figure 1. This methodology is somewhat different in that we design general components of our system before actually defining the system itself.

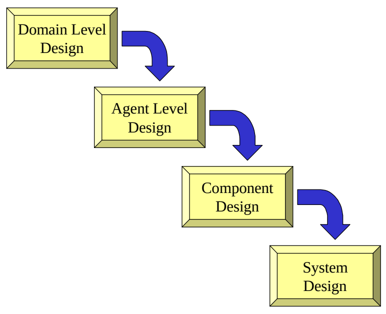

Figure 1. MaSE Methodology

2.1. Domain Level Design

The first step in MaSE is *domain level design*, which captures the basic types and interactions between agents in our system. At this level, whether or not an agent has intelligence, how that intelligence is captured, or how the agent is defined is not important. We are concerned with only the high-level definition of the types of agents, their goals, and their external interfaces. It is these external interfaces that define coordination protocols between the agents. We define how each agent may coordinate with other agents and which agent types they may need to coordinate with. We leave the actual numbers, locations, and specific responsibilities of the actual agents within our system until system design. The steps in domain-level design are:

1. Identifying agent types.
2. Identifying the possible interactions between agent types.
3. Defining coordination protocols for each type of interaction.

Step 1, identifying agent types, is analogous to object oriented techniques where we look at the problem space to determine the types of agents needed to effectively model the domain. While MaSE does not currently define a specific technique for agent type identification, the use of role modeling [5, 15], use cases, and collaboration diagrams [8] have proven to be useful. Once we have identified the types of agents, we identify possible interactions that might occur between different types of agents. These interactions become agent conversations that are defined using coordination protocols. These protocols describe the possible sequences of messages that may be passed between agents to achieve coordination. The domain level design is captured using various AgML diagrams that are discussed in Section 3 below.

2.2. Agent Level Design

The next step in MaSE is the *agent level design*. It is at this level that we define (or reuse) the agent architectures for each individual agent type. The agent architecture defines the components within each agent and how they interact. The agent level is documented using the AgDL. Specific design steps at the agent level are:

1. Mapping actions identified in agent conversations to internal components.
2. Defining data structures identified in agent conversations. These data structures represent input or output from the agent.
3. Defining additional data structures, internal to the agent. These data structures represent data flows between components in the architecture.

2.3. Component Design

The *component design* level is the next obvious level of the MaSE methodology. Once the agent architecture is defined, the components specified must be designed.

Components being designed from scratch are defined using AgDL. However, if components exist that can be re-used, it is our hope that we can define agents in such a way as to take advantage of existing component enabling technologies, such as JavaBeans, to allow us to reuse many components. Obvious agent components include planners, inference mechanisms, search algorithms, and learning algorithms.

2.4. System Design

Finally, *system design* takes place once the design of the domain, agents, and components are complete. By defining the domain first, system design becomes an exercise in picking the number and types of agents needed as well as defining specified parameters within the agent definition. Although not technically part of system design, once a system has been defined we can verify certain properties of interest such as safety and liveness. The overall system design is specified using an AgML deployment diagram. Specific steps in system design include:

1. Selecting the agent types that are needed.
2. Determining the number of agents required of each type and defining:
   a. The agent's physical location or address.
   b. The types of conversations that agents will be able to hold.
   c. Any other parameters defined in the domain.

Defining a domain and system requires a set of formal tools to be able to reuse existing components, synthesize new components, and analyze various properties of the system. Both AgML and AgDL combine to provide a formal definition suitable for software synthesis or component reuse. We discuss the specific AgML diagrams and their uses in the next section.

3. Agent Modeling Language

The AgML uses four diagrams to define high-level features of multiagent systems. The first three diagrams define the domain model used to develop the system. This domain model describes how various agent types are related and how the agents can be combined to form multiagent systems. First, an *Agent Diagram* is used to define the types of agents in the system and the possible communications paths, defined as conversations, between agents in the system. The *Communication Hierarchy Diagram* defines the relationship of the various classes of conversations within the system. Many conversation types are the same, or slight variants of each other. The *Communication Class Diagram* is a state machine based representation used to define conversations between two or more agents. Conversations define the legal sequences of messages that can be sent between two agents involved in a conversation. These conversations also tie the messages and their data to internal components of the agent. Finally, *Deployment Diagrams* are used to define the actual system being developed. Deployment Diagrams define the actual number, location, and types of agents in the system.

3.1. Agent Diagrams

The classes of agents in a particular domain are described using *Agent Diagrams,* which are very similar in look and use to many object diagrams. The major differences between agent diagrams and object diagrams are (1) the interfaces to the agents and (2) the semantics of the relationships between agents. Basically agent interfaces are defined by the set of services they can provide while the relationships between agents define conversations that can be held between agents. Thus an Agent Diagram defines the existence of agent classes and their relationships in the logical view of the system. An example of an agent class definition is shown in Figure 2.

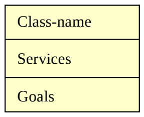

Figure 2. Agent Class

Each class of agents belongs to an *agent class* that is much like an object-oriented class. The difference lies in the interface to the agent. In object-oriented classes, each class has attributes and methods. External objects may, if allowed, look at the object's attribute values and invoke its methods. In the agent abstraction, each agent has a goal and may or may not provide services to other agents. For instance, an information agent may monitor a particular web page waiting for it to change. When the web page changes, it may analyze the change and send updates to those registered agents who might be affected by the change. Such an agent might be defined as shown in Figure 3.

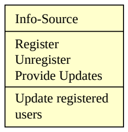

Figure 3. Information Agent Class

In order to provide services, an agent must interact with other agents in the system. In AgML, interactions between agents take place when agents have *conversations*. Barbuceanu and Fox [1] use conversations to define *coordination protocols* between agents. These conversations define a shared convention about message exchanges that the agents use to coordinate their actions and are defined using communication class diagrams. Often one agent is requesting an advertised service of another agent. In order for the agents to communicate appropriately, coordination protocol must be established between them. In object-orientation, relationships between object classes are described abstractly using *associations*. In an agent diagram, these associations are more concrete – they are *conversations*. Thus, for the agent described above, the conversations that might be required are shown in Figure 4.

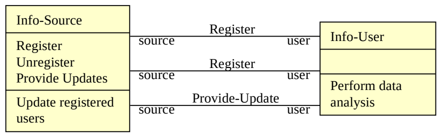

Figure 4. Conversations

The labeled lines between the two agents denote *conversations*. The name of the conversation is above and centered between the two agent classes. The labels below and next to each agent class are the names of the *roles* the agents play in the conversation. It is possible for two or more agent classes to be involved in the same conversation classes and thus the role name defines which part which agent class plays in the conversation.

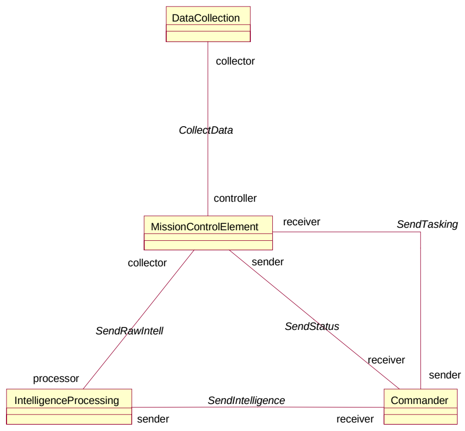

Figure 5. Agent Diagram

For instance, in the agent diagram of Figure 5, the *SendRawIntell*, *SendIntelligence*, *SendStatus*, and *SendTasking* conversations are all based on the same conversation type, *SendInfo*. They were renamed to provide a clearer context, but could have been all been labeled the same as shown in Figure 6.

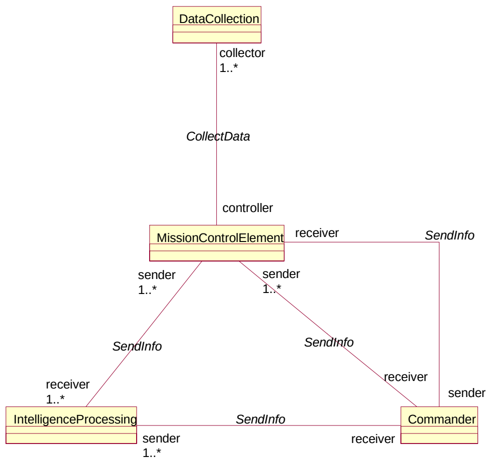

Figure 6. Agent Diagram with Reused Conversations

Each conversation defined in the Agent Diagram has a corresponding class in the Communication Hierarchy Diagram and Communication Class Diagram, which are discussed in depth below. Each side of a conversation also has a notation describing the multiplicity of the relationships between agents. For instance, in Figure 6, the collector side of the CollectData conversation is annotated with the multiplicity *1..** while the controller side of the conversation has no annotation. This basically means that each controller (MissionControlElement agent) may have conversations with one or more collectors (DataCollection agents) but that each collector may only converse with a single controller. In general, multiplicities may be specified by a sequence of integer intervals in the format

lower-bound .. upper-bound

with each interval separated by commas. In addition, the asterisk (*) may be used to indicate and unlimited upper bound. The default interval is equivalent to *1..1*, which denotes exactly one as in the case of the MissionControlElement agent.

3.2. Communication Hierarchy Diagrams

Communication Hierarchy Diagrams define the relationships between the various conversations within a system. For instance, as described above, the *Send* conversations of Figure 5 are all specializations of the *SendInfo* conversation as shown in Figure 7. There are basically two types of conversation protocols within the system: CollectData and SendInfo. The SendRawIntell, SendIntelligence, SendStatus, and SendTasking conversations are all subtypes of the basic SendInfo conversation. The Communication Hierarchy Diagram simply defines the relationships between conversations; the conversations themselves are described using Communication Class Diagrams. Some specializations that seem appropriate include simple renaming, modifying data types, and adding additional states and messages that do not violate the generalized conversation definition. The usefulness of the last specialization is under investigation.

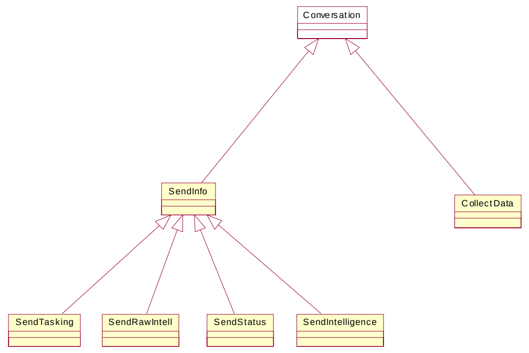

Figure 7. Communication Hierarchy Diagram

3.3. Communication Class Diagrams

Communication Class Diagrams are a set of finite state machine (FSM) diagrams that define the conversation states that each agent may be in as defined by the agent's role in the conversation. Each side of a conversation is defined by a separate FSM, although two FSM diagrams are needed to completely define a Communication Class Diagram for a binary conversation. For instance, Figure 8 defines the FSM for the receiver side of the SendInfo conversation. The labeling on the arcs follows conventional UML notation shown below.

rec-mess(args1)[cond] ^ *trans-mess(args2)*

This notation states that if the message, *rec-mess,* is received with arguments, *args1,* and the condition, *cond,* holds, then the message, *trans-mess,* is transmitted with arguments *args2.* The conversation then transitions from the starting state to the state pointed at by the associated translation arrow. If the conversation in Figure 9 is in the *wait* state and a *failure-transmission* message is received, the agent sends a *send* message with *information* as an argument. In this particular case, the conversation transitions back to the *wait* state since the transition arc is a loop.

Conversations start when one agent sends a message to another agent. If the message does not match an existing conversation and there is a transition from the start state, ●, matching the message, a new conversation is started. The label on the transition from the *start* state defines the affect of the transition as defined above. Once in a non-start state, the conversation waits until a transition occurs that takes it to a new state. For instance, in Figure 9, if the conversation is in the *Wait*, receipt of an *acknowledge* or *failure-transmission* message will cause the conversation to transition to the end state or back to the wait state respectively. Once the conversation transitions to the end state, the conversation is completed.

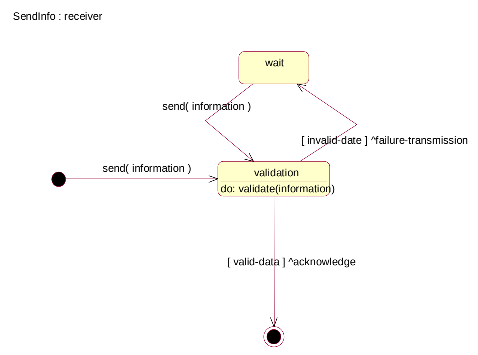
SendInfo : receiver
wait
send(information)
[invalid-date] ^failure-transmission
send(information)
validation
do: validate(information)
[valid-data] ^acknowledge

Figure 8. SendInfo Conversation -- Receiver Role

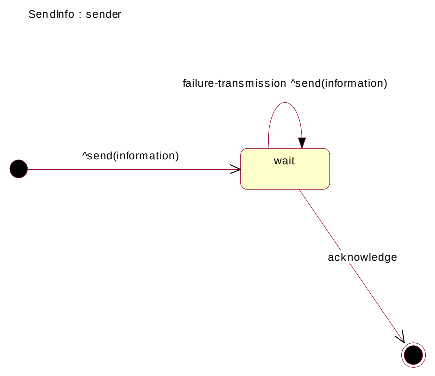

Figure 9. SendInfo Conversation -- Sender Role

While in the *validation* state in Figure 8, the *do:* construct specifies that the agent is doing the action *validate(information).* This is how we link the conversations defined in Communications Diagrams to processes within the agent. The fact that the only two transitions out of the *validation* state are *null* transitions (there is no incoming message that forces the transition) specifies that the conversation remains in this state until the *validation* procedure is completed. Then, based on whether a *valid-data* or *invalid-data* is output from the *validation* procedure, the conversation transitions to the *end* state or remains in the *validation* state.

The semantics of the *information* argument in Figure 9 is dependent on the agents involved and the purpose of the conversation. That is why we often specialize basic conversations to meet the specific needs of two interacting agents and to clarify the actual purpose of the conversations. Figure 10 and Figure 11 show the *SendRawIntell* conversation, which is a specialization of the *SendInfo* conversation.

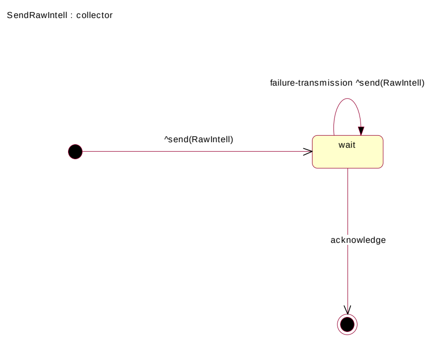
SendRawIntell : collector
failure-transmission ^send(RawIntell)
^send(RawIntell)
wait
acknowledge

Figure 10. SendRawIntell -- Sender Role

In this case, the only real specialization is the renaming of *information* to *RawIntell*. Obviously, RawIntell is information, but we have specialized it to make the semantics clearer. Also, since the Mission Control Element is involved with two separate types of SendInfo conversations, specialization makes it clear that SendRawIntell conversations are held with IntelligenceProcessing agents while SendStatus conversations are held with COMMANDER agents. Figure 11 shows the receiver side of the conversation. This shows that both sides of the conversation have to be specialized so that they can both work together.

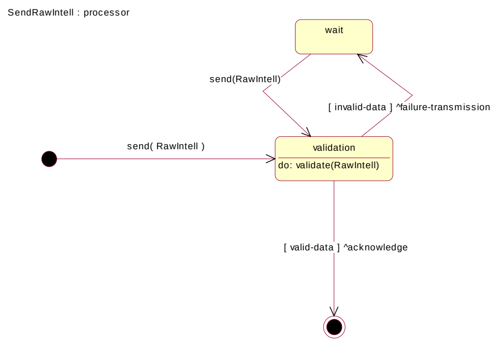
SendRawIntell : processor
wait
send(RawIntell)
[invalid-data] ^failure-transmission
send(RawIntell)
validation
do: validate(RawIntell)
[valid-data] ^acknowledge

Figure 11. SendRawIntell -- Receiver Role

Figure 12 and Figure 13 show both sides of the CollectData conversation between the controller (MissionControlElement agents) and the collector (DataCollection agents).

The controller starts this conversation by sending a *collect-data* message with *sensor* and *location* arguments. Upon receiving the message, the collector moves to the *collecting* state. The actions specified within the state represent processing required by the agent. In this case, upon *entry* into the *collecting* state, the agent performs the *move-to(location)* process. When that process is complete, the agent performs *collect-data* process with the argument *sensor*. When this process is complete, one of its three outputs determines the next state: *collection-complete*, *sensor-failure*, or *movement-failure*. The three transitions that occur on these outputs are null transitions, that is, they do not require an incoming event to cause the transition.

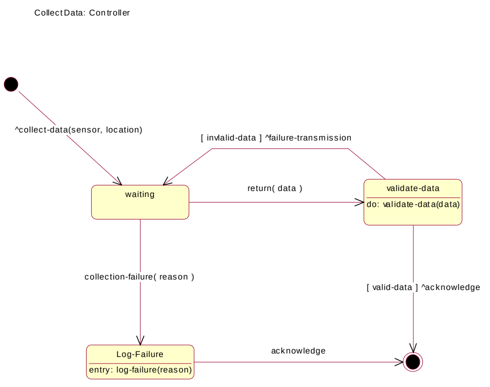
CollectData: Controller
^collect-data(sensor, location)
[invlalid-data] ^failure-transmission
waiting
return(data)
validate-data
do: validate-data(data)
collection-failure(reason)
[valid-data] ^acknowledge
Log-Failure
entry: log-failure(reason)
acknowledge

Figure 12. Collect Data Conversation Diagram for Controller

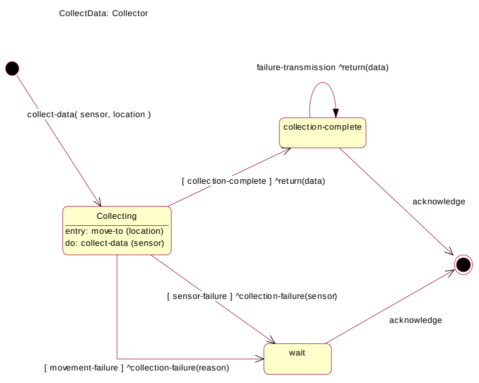
CollectData: Collector
failure-transmission ^return(data)
collect-data(sensor, location)
collection-complete
[collection-complete] ^return(data)
acknowledge
Collecting
entry: move-to (location)
do: collect-data (sensor)
[sensor-failure] ^collection-failure(sensor)
acknowledge
wait
[movement-failure] ^collection-failure(reason)

Figure 13. Collect Data Conversation Diagram for Collector

It is worth noting that the *move-to(location)* action could take a considerable amount of time to complete. Thus, conversations must be able to be completed over long periods. This also emphasizes the need for agents to be able to be involved with more than one conversation at a time. If the controller had to wait for hours for one of its collectors to move to the appropriate location to collect data, it would be unable to control its other collectors. It would be possible for the collector to incorporate sending a *status* message every few minutes to keep its controller informed of its current progress and condition.

3.4. Deployment Diagrams

Deployment Diagrams are used to define a system based on agents defined in the Domain Level, Agent Level, and Component Level design steps. Deployment Diagrams define system parameters such as the actual number, types, and locations of the agents in the system. Figure 14 shows an example deployment diagram for the domain defined in Figure 5.

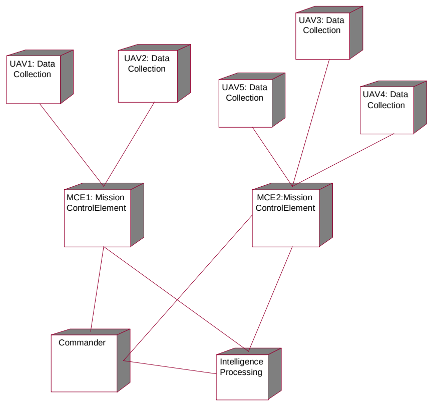

Figure 14. Deployment Diagram

4. Discussion

4.1. Formal Semantics

Although not discussed in depth in this paper, both AgML and AgDL semantics are based of multi-sorted algebras. Algebraic approaches have the advantage that there has been a great deal of work in automatically synthesizing code from algebraic specifications [12, 6]. Other formal specification languages such as Z and VDM, while providing a formal semantics, do not provide the same strong notion of refinement from specification to code.

An example of our formalization is shown in Figure 15 and Figure 16. These specifications formalize each side of the conversations defined graphically in Figure 8 and Figure 9. Our formalization uses algebraic specifications to define abstract data types that define the effect of incoming messages on a specific conversation. These specifications can be combined, using category theoretic approaches, and used as the basis for proving theorems about the specific conversation as well as the system as a whole. Also, given the formal semantics and a special purpose reasoning system, we can verify general properties of interest such as safety and liveness. For example, we will want to know under what conditions a particular conversation will actually terminate. In the example of Figure 15 and Figure 16, analysis reveals that if at some point, *validate(i) = valid-data*, then the conversation will end successfully.

```
spec sendinfo-sender
  sorts
       conversation, state, info
  operations
       new :  → conversation
    % attributes
       state : conversation → state
    % messages
       failure-transmission : conversation → boolean
       acknowledge : conversation → boolean
       send : conversation , info → boolean
    % states
       start : → state
       end : → state
       wait : → state
  axioms
       new() = c ⇒  state(c) = start
       state(c) = start ⇒ state(c) = wait ^ send(c, i)
       state(c) = wait ^ failure-transmission(c) ⇒ send(c, i)
       state(c) = wait ^ acknowledge(c) ⇒ state(c) = end
end spec
```

Figure 15. SendInfo - Sender Specification

4.2. Practical Benefits

The similarity of MaSE to existing object-oriented methodologies provides other benefits. Although MaSE and AgML support formal multiagent system synthesis, the graphical nature of the AgML makes it easier to learn, use, and understand than other textually based formal representations. Also, the look and feel of AgML is similar to many object-oriented methodologies making it more natural for persons with object-oriented experience. In effect, our work with MaSE is an attempt to build formally based multiagent systems without the designer realizing that he or she is really using formal methods.

MaSE and AgML together provide many advantages over traditional software engineering techniques. These advantages include:

1. A higher level of abstraction than traditional systems engineering techniques. In particular, agents are a higher level abstraction of objects from object-oriented software engineering. Because of this abstraction, MaSE can capture traditional object-oriented systems as well as agent-based systems for which traditional techniques are inappropriate.

2. A more concise representation than object-oriented techniques. For instance, a formal definition of traditional object oriented *associations* is generally not possible because the use of associations is not standardized. This lack of precise and formal usage in traditional, informal techniques makes them difficult to use for software synthesis.

3. A formal syntax and semantics. This advantage supports automated software synthesis and component reuse and allows properties of interest to be more easily proved.

```
spec sendinfo-receiver
  sorts
        conversation, state, info, result
  operations
        new : → conversation
    % attributes
        state : conversation → state
    % messages
        failure-transmission : conversation → boolean
        acknowledge : conversation → boolean
        send : conversation, info → boolean
    % agent operations
        validate : info → result
    % states
        start : → state
        end : → state
        wait : → state
        validation : → state
 axioms
        new() = c ⇒  state(c) = start
        state(c) = start ^ send(c, i) ⇒ state(c) = validation
        state(c) = validation ^ validate(i) = invalid-data
                                    ⇒ state(c) = wait  ^ failure-transmission(c)
        state(c) = wait ^ send(c, i) ⇒ state(c) = validation
        state(c) = validation ^ validate(i) = valid-data
                                    ⇒ state(c) = end ^ acknowledge(c)
end spec
```

Figure 16. SendInfo - Receiver Specification

4.3. Related Work

Our work is similar in many respects to the agent methodologies based on object-oriented concepts [15, 4]. However, few of these have a formal basis. Some work in formalization of agent systems has been performed in Z [7, 3] but has focused on formal modeling and not automated code synthesis.

4.4. Future Work

This paper represents our initial attempts to define a methodology and system modeling language for multiagent systems. Our goal is to integrate this methodology and language into an automated multiagent system synthesis system called agentTool, which will formally verify and generate multiagent systems that are correct by construction. agentTool will provide a graphical user interface based on AgML to design and verify multiagent systems. Ongoing work includes formal definition of AgML semantics as well as a complete description of AgDL that is seamlessly integrated with AgML in agentTool. We are also investigating appropriate methods for verifying interesting properties of multiagent systems. This research requires formal definition of the underlying communications frameworks that implement the conversations defined in AgML as well as the conversation protocols themselves. Future work includes the definition and use of predefined components within agentTool.

5. Acknowledgements

This research was sponsored by a grant from the Air Force Office of Scientific Research.

Bibliography

1. Barbuceanu, M. and Fox M. 1995. COOL: A Language for Describing Coordination in Multi Agent Systems. *Proceedings of the First International Conference on Multi-Agent Systems*, AAA Press/The MIT Press, June 1995, 17-25.
2. DeLoach, S. A. and Hartrum, T. C. 1999. A Theory-Based Representation for Object-Oriented Domain Models. Accepted for publication in *IEEE Transactions on Software Engineering*.
3. d'Inverno, M., Kinny, D., Luck, M. & Wooldridge M. 1997. A Formal Specification of dMARS. In *Intelligent Agents IV (LNAI Vol 1365): 155-176.* Springer-Verlag, Berlin.
4. Iglesias, C.A., Garijo, M., & Gonzalez, J.C. 1999. A Survey of Agent-Oriented Methodologies. In *Intelligent Agents V – Proceedings of the Fifth International Workshop on Agent Theories, Architectures, and Languages (ATAL-98)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Heidelberg.
5. Kendall E. A. 1998. Agent Roles and Role Models. Intelligent Agents for Information and Process Management (AIP'98).
6. Kestrel Institute 1994. *Specware User manual: Specware Version Core4*. October 1994.
7. Kinny, D., Georgeff, M., & Rao, A. 1996. A Methodology and Modelling Technique for Systems of BDI agents. In *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent*

World, (LNAI Vol 1038): 56-71. Springer-Verlag, Berlin.

8. Muller, P. 1997. *Instant UML*. Wrox Press, Birmingham, UK.

9. Nwana, H. S. 1996. Software Agents: An Overview. *Knowledge Engineering Review* 11(3): 205-244.

10. Pont, M. J., and Moreale, E. 1996. Towards a Practical Methodology for Agent-Oriented Software Engineering with C++ and Java. Technical Report, TR 96-33, Department of Engineering, Leicester University.

11. Rumbaugh, J. et. al. 1991. *Object-Oriented Modeling and Design*. Prentice-Hall, Inc. Englewood Cliffs, New Jersey.

12. Smith, D. R. 1990, KIDS – A Semi-automatic Program Development System. *IEEE Transactions on Software Engineering* 16(9): 1024-1043.

13. Sycara, K. P. 1998, Multiagent Systems. *AI Magazine* 19(2): 79-92.

14. Wooldridge, M., & Jennings, N. 1995. Intelligent Agents: Theory and Practice. *Knowledge Engineering Review,* 10(2): 115-152.

15. Wooldridge, M., Jennings, N., & Kinny, D. 1999. A Methodology for Agent-Oriented Analysis and Design. *to be presented at Agents'99, Seattle WA, May 1999.*