Intelligent Agents VII. Agent Theories, Architectures, and Languages - 7th. International Workshop, ATAL-2000, Boston, MA, USA, July 7-9, 2000, Proceedings, Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, 2001.

Developing Multiagent Systems with agentTool

Scott A. DeLoach and Mark Wood

Department of Electrical and Computer Engineering
Air Force Institute of Technology
2950 P Street, Wright-Patterson AFB, OH 45433-7765
scott.deloach@afit.edu

Abstract. The advent of multiagent systems has brought together many disciplines and given us a new way to look at intelligent, distributed systems. However, traditional ways of thinking about and designing software do not fit the multiagent paradigm. This paper describes the Multiagent Systems Engineering (MaSE) methodology and agentTool, a tool to support MaSE. MaSE guides a designer from an initial system specification to implementation by guiding the designer through a set of inter-related graphically based system models. The underlying formal syntax and semantics of clearly and unambiguously ties them together as envisioned by MaSE.

1 Introduction

The advent of multiagent systems has brought together many disciplines in an effort to build distributed, intelligent, and robust applications. They have given us a new way to look at distributed systems and provided a path to more robust intelligent applications. However, many of our traditional ways of thinking about and designing software do not fit the multiagent paradigm. Over the past few years, there have been several attempts at creating tools and methodologies for building such systems. Unfortunately, many of the methodologies have focused on single agent architectures [9, 13] or have not been adequately supported by automated toolsets [5, 17]. In our research, we have been developing both a complete-lifecycle methodology and a complimentary environment for analyzing, designing, and developing heterogeneous multiagent systems. The methodology we are developing is called Multiagent Systems Engineering (MaSE) while the tool we are building to support that methodology is called agentTool.

In this research, we view agents as a specialization of the objects. Instead of objects whose methods that are invoked directly 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 can handle intelligent and non-intelligent system components equally within the same framework. This view also justifies our use object-oriented tools and techniques. Since agents are specializations of objects, we can tailor general object-oriented methods and apply them to the specification and design of multiagent systems.

2 Multiagent Systems Engineering Methodology

The general flow of MaSE follows the seven steps shown in Figure 1. The rounded rectangles on the left side denote the models used in each step. The goal of MaSE is to guide a system developer from an initial system specification to a multiagent system implementation. This is accomplished by directing the designer through this set of inter-related system models. Although the majority of the MaSE models are graphical, the underlying semantics clearly and unambiguously defines specific relationships between the various model components.

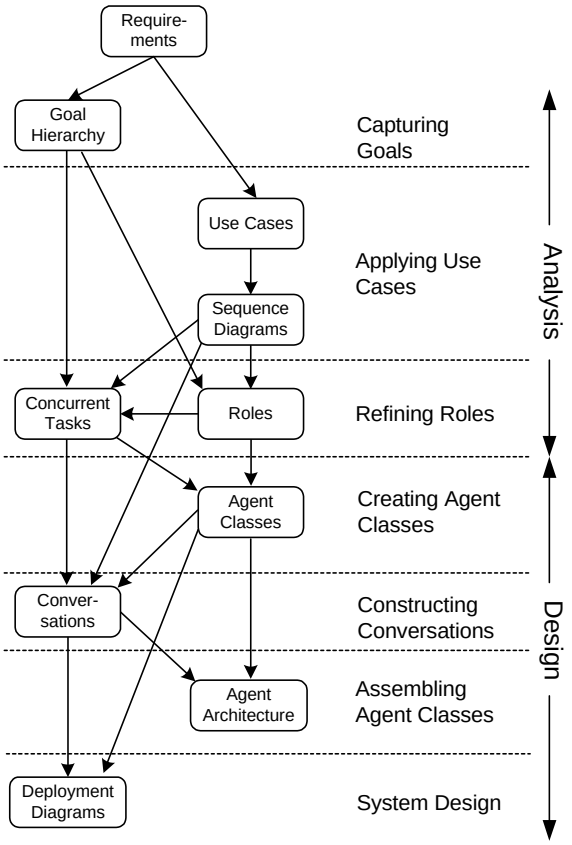

Fig. 1. MaSE Methodology

MaSE is designed to be applied iteratively. Under normal circumstances, we would expect a designer to move through each step multiple times, moving back and forth between models to ensure each model is complete and consistent. While this is common practice using most design methodologies, MaSE was specifically designed to support this process by formally capturing the relationships between the models. By automating the MaSE models in our agentTool environment, these relationships are captured and enforced thus supporting the designer's ability to freely move

between steps. The result is consistency between the various MaSE models and a system design that satisfies the original system goals.

MaSE, as well as agentTool, is independent of a particular multiagent system architecture, agent architecture, programming language, or communication framework. Systems designed using MaSE can be implemented in a variety ways. For example, a system could be designed and implemented that included a heterogeneous mix of agent architectures and used any one of a number of existing agent communication frameworks. The ultimate goal of MaSE and agentTool is the automatic generation of code that is correct with respect to the original system specification. With such a capability, MaSE and agentTool could work hand in hand with flexible and efficient runtime environments such as JADE [1].

2.1 Capturing Goals

The first step in the MaSE methodology is *Capturing Goals*, which takes the initial system specification and transforms it into a structured set of system goals, depicted in a *Goal Hierarchy Diagram*, as shown in Figure 2. In MaSE, a *goal* is always defined as a system-level objective. Lower-level constructs may inherit or be responsible for goals, but goals always have a system-level context.

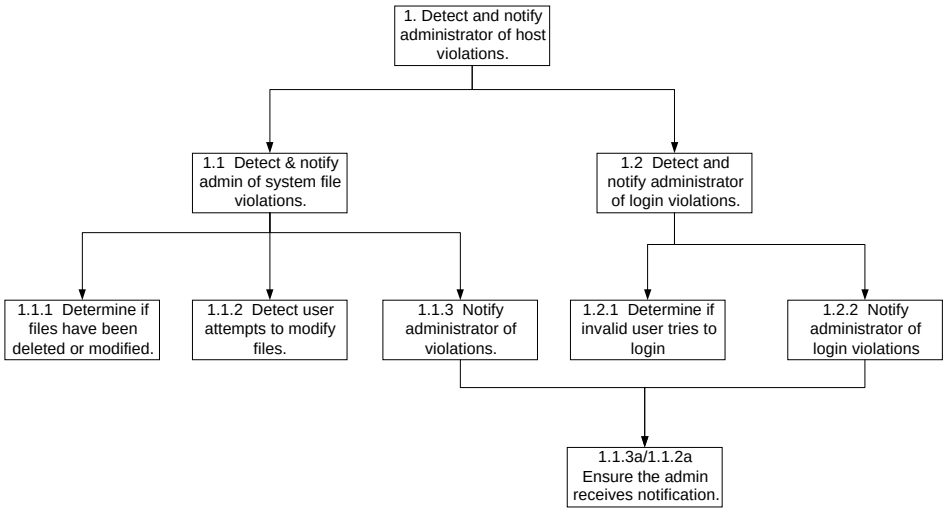

Fig. 2. Goal Hierarchy Diagram

There are two steps to *Capturing Goals*: identifying the goals and structuring goals. Goals are identified by distilling the essence of the set of requirements. These requirements may include detailed technical documents, user stories, or formalized specifications. Once these goals have been captured and explicitly stated, they are less likely to change than the detailed steps and activities involved in accomplishing them [7]. Next, the identified goals are analyzed and structured into a Goal Hierarchy Diagram. In a Goal Hierarchy Diagram, goals are organized by importance. Each

level of the hierarchy contains goals that are roughly equal in scope and sub-goals are necessary to satisfy parent goals. Eventually, each goal will be associated with roles and agent classes that are responsible for satisfying that goal.

2.2 Applying Use Cases

The *Applying Uses Cases* step is a crucial step in translating goals into roles and associated tasks. U*se cases* are drawn from the system requirements and are narrative descriptions of a sequence of events that define desired system behavior. They are examples of how the system should behave in a given case.

To help determine the actual communications required within a multiagent system, the use cases are restructured as Sequence Diagrams, as shown in Figure 3. A *Sequence Diagram* depicts a sequence of events between multiple roles and, as a result, defines the minimum communication that must take place between roles. The roles identified in this step form the initial set of roles used to fully define the system roles in the next step. The events identified here are also used later to help define tasks and conversations since all events between roles will require a conversation between the agent classes if the roles are played by different agent classes.

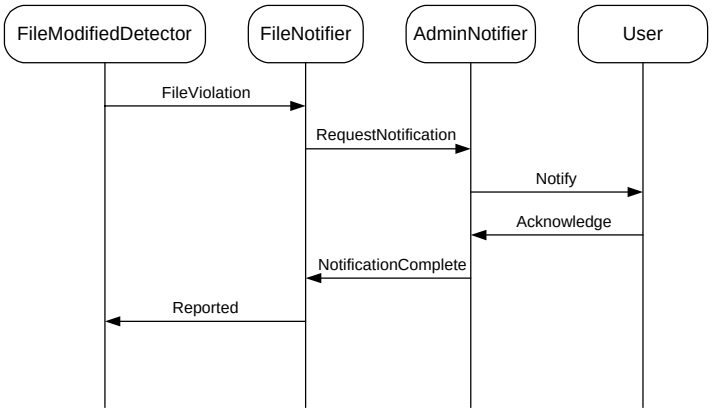

Fig. 3. Sequence Diagram

2.3 Refining Roles

The third step in MaSE is to ensure we have identified all the necessary roles and to develop the tasks that define role behavior and communication patterns. Roles are identified from the Sequence Diagrams developed during the Applying Use Cases step as well as the system goals defined in Capturing Goals. We ensure all system goals are accounted for by associating each goal with a specific role that is eventually played by at least one agent in the final design. A *role* is an abstract description of an entity's expected function and is similar to the notion of an actor in a play or an office within an organization [6]. Each goal is usually mapped to a single role. However, there are many situations where it is useful to combine multiple goals in a single role

for convenience or efficiency. We base these decisions on standard software engineering concepts such as functional, communicational, procedural, or temporal cohesion. Other factors include the natural distribution of resources or special interfacing issues. Roles are captured in a Role Model as shown in Figure 4.

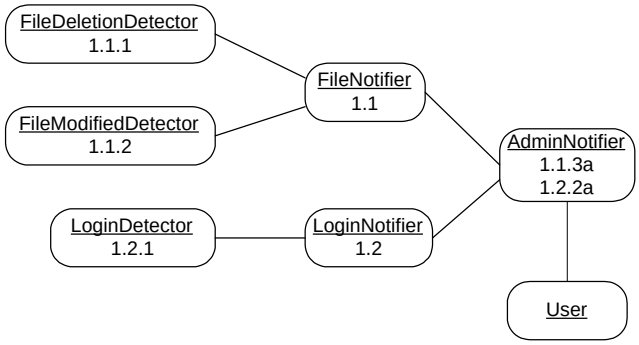

Fig. 4. Role Model

Once roles have been defined, tasks are created. A set of concurrent tasks provide a high-level description of what a role must do to satisfy its goals including how it interacts with other roles. An example of a MaSE *Concurrent Task Diagram*, which defines the Notify User task of the AdminNotifier role, is shown in Figure 5. The syntax of a transition follows the notation shown below [3].

```
trigger(args1) [ guard ] / transmission(args2)
```

The statement is interpreted to say that if an event *trigger* is received with a number of arguments *args*1 and the condition *guard* holds, then the message *transmission* is sent with the set of arguments *args2*. All items are optional. For example, a transition with just a guard condition, *[guard]*, is allowed, as well as one with a received message and a transmission, *trigger / transmission*. Multiple transmission events are also allowed and are separated by semi-colons (;). Actions may be performed in a state and are written as functions.

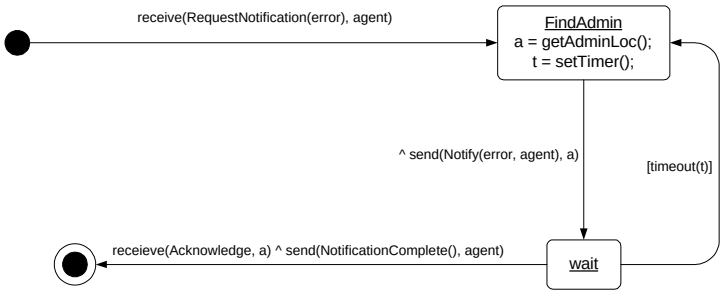

Fig. 5. MaSE Task

2.4 Creating Agent Classes

In *Creating Agent Classes*, agent classes are identified from roles and documented in an Agent Class Diagram, as shown in Figure 6. Agent Class Diagrams depict agent classes as boxes and the conversations between them as lines connecting the agent classes. As with goals and roles, we generally define a one-to-one mapping between roles, which are listed under the agent class name, and agent classes. However, the designer may combine multiple roles in a single agent class or map a single role to multiple agent classes. Since agents inherit the communication paths between roles, any paths between two roles become conversations between their respective classes. Thus, as the designer assigns roles to agent classes, the overall organization of the system is defined. To make the organization more efficient, it is often desirable to combine two roles that share a high volume of message traffic. When determining which roles to combine, concepts such as cohesion and the volume of message traffic are important considerations.

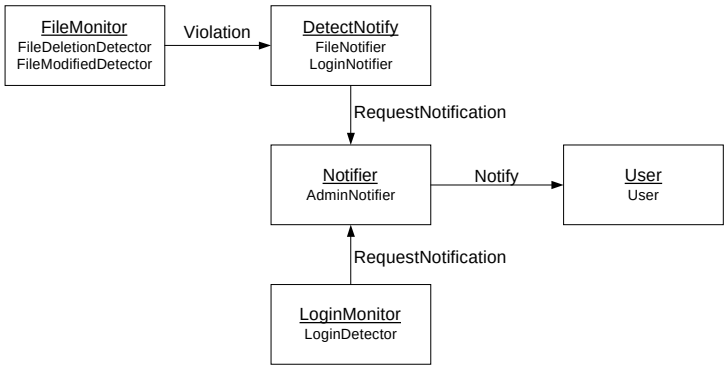

Fig. 6. Agent Class Diagram

2.5 Constructing Conversations

Constructing Conversations is the next step of MaSE, which is often performed almost in parallel with the succeeding step of Assembling Agents. The two steps are closely linked, as the agent architecture defined in Assembling Agents must implement the conversations and methods defined in Constructing Conversations. A MaSE conversation defines a coordination protocol between two agents. Specifically, a conversation consists of two Communication Class Diagrams, one each for the initiator and responder. A *Communication Class Diagram* is a pair of finite state machines that define a conversation between two participant agent classes. One side of a conversation is shown in Figure 7. The initiator always begins the conversation by sending the first message. The syntax for Communication Class Diagrams is very similar to that of Concurrent Task Diagrams. The main difference between conversations and concurrent tasks is that concurrent tasks may include multiple conversations between many different roles and tasks whereas conversations are binary exchanges between individual agents.

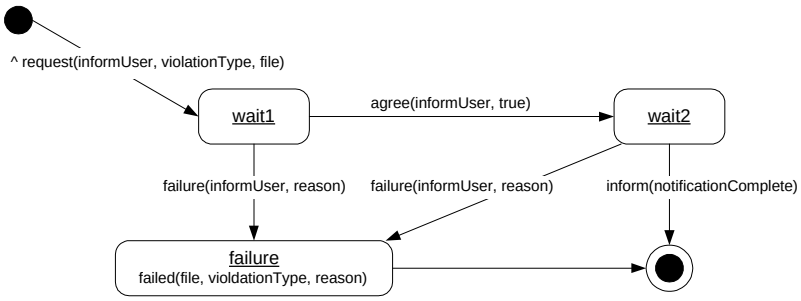

Fig. 7. Communication Class Diagram

2.6 Assembling Agents

In this step of MaSE, the internals of agent classes are created. Robinson [15] describes the details of assembling agents from a set of standard or user-defined architectures. This process is simplified by using an architectural modeling language that combines the abstract nature of traditional architectural description languages with the Object Constraint Language, which allows the designer to specify low-level details. When combined, these two languages provide the same capabilities as the text based architectural language of HEMASL found elsewhere in this volume [11]. A current research focus is how to map tasks to conversations and internal agent architectures. The actions specified in the tasks and conversations must be mapped to internal functions of the agent architecture.

2.7 System Deployment

The final step of MaSE defines the configuration of the actual system to be implemented. To date, we have only looked at static, non-mobile systems although we are currently investigating the specification and design of dynamic and mobile agent systems. In MaSE, we define the overall system architecture using Deployment Diagrams to show the numbers, types, and locations of agents within a system as shown in Figure 8. The three dimensional boxes denote individual agents while the lines connecting them represent actual conversations. A dashed-line box encompasses agents that are located on the same physical platform.

The agents in a Deployment Diagram are actual instances of agent classes from the Agent Class Diagram. Since the lines between agents indicate communications paths, they are derived from the conversations defined in the Agent Class Diagram as well. However, just because an agent type or conversation is defined in the Agent Class Diagram, it does not necessarily have to appear in a Deployment Diagram.

System Deployment is also where all previously undefined implementation decisions, such as programming language or communication framework, must be made. While in a pure software engineering sense, we want to put off these decisions until this step, there will obviously be times when the decision are made early, perhaps even as part of the requirements.

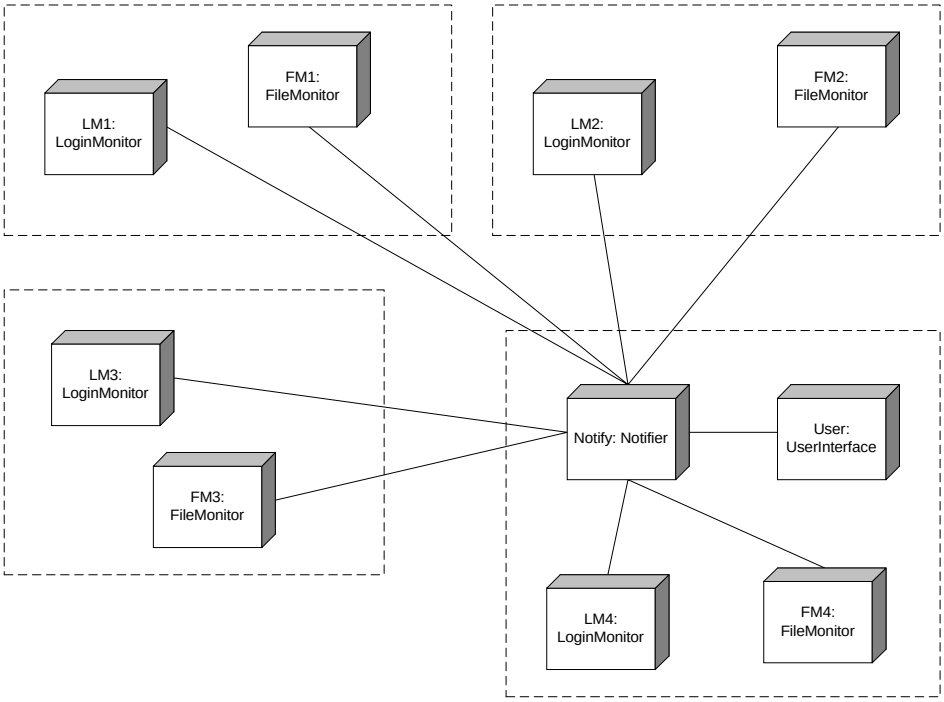

Fig. 8. Deployment Diagram

3 agentTool

The agentTool system is our attempt to implement a tool to support and enforce MaSE. Currently agentTool implements three of the seven steps of MaSE: Creating Agent Classes, Constructing Conversations, and Assembling Agent Classes. We are adding support for the analysis phase.

The agentTool user interface is shown in Figure 9. The menus across the top allow access to several system functions, including a persistent knowledge base [14], conversation verification [10], and code generation. The buttons on the left add specific items to the diagrams while the text window below them displays system messages. The different MaSE diagrams are accessed via the tabbed panels across the top of the main window. When a MaSE diagram is selected, the designer can manipulate it graphically in the window. Each panel has different types of objects and text that can be placed on them. Selecting an object in the window enables other related diagrams to become accessible. For example, in Figure 9, two agents have been added with a conversation between them. When the user selects the *Advertise* conversation (by clicking on the line), the *Conv:Advertise Initiator* and *Conv:Advertise Responder* tabbed panes become visible. The user may then access those diagrams by selecting the appropriate tab.

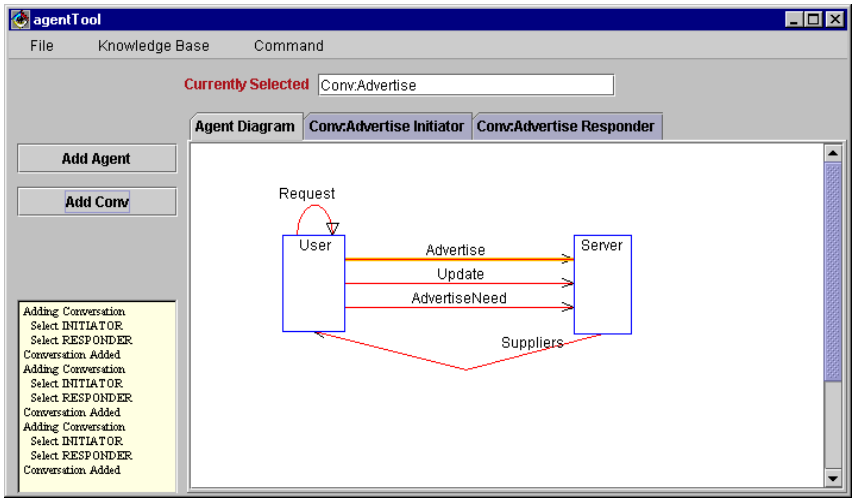

Fig. 9. Agent Class Diagram Panel

The part of agentTool that is perhaps the most appealing is the ability to work on different pieces of the system and at various levels of abstraction interchangeably, which mirrors the ability of MaSE to incrementally add detail. The "tabbed pane" operation of agentTool implements this capability of MaSE since the step you are working on is always represented by the current diagram and the available tabs show how you might move up and down through the methodology.

It is easier to envision the potential of this capability by considering the implementation of the entire MaSE methodology in agentTool. During each step of system development, the various analysis and design diagrams would be available through tabs on the main window. The ordering of the tabs follows the MaSE steps, so selecting a tab to the left of the current pane would move "back" in the methodology while selecting a tab to the right would move "forward." The available diagrams (tabs) are controlled by the currently selected object. The available diagrams include those that can be reached following valid MaSE steps. For instance, by selecting a conversation, tabs for the associated Communication Class Diagrams become available while selecting an agent would cause a tab for the Agent Architecture Diagram to appear.

3.1 Building a Multiagent System using agentTool

Constructing a multiagent system using agentTool begins in an Agent Class Diagram as shown above in Figure 9. Since a conversation can only exist between agent classes, agent classes are generally added before conversations. While we can add all the agent classes to the Agent Class Diagram before adding any conversations, we can also add "sections" of the system at a time, connecting appropriate agent classes with conversations, and then moving onto the next section. Either method is supported and is generally a matter of personal choice.

3.2 Constructing Conversations in agentTool

Once we have defined agent classes and conversations, we can define the details of the conversations using Communication Class Diagrams. The "Add State" button adds a state to the panel while the "Add Conversation" button adds a conversation between the two selected states. A conversation can be verified at any point during its creation by using the *Verify Conversations* command from the Command menu [10]. The agentTool verification process ensures conversation specifications are deadlock free. If any errors exist, the verification results in a highlighted piece or pieces of a conversation, as shown in Figure 10 on the "Ack" transition (highlights are yellow in the application). Each highlight indicates a potential error as detected by the verification routine.

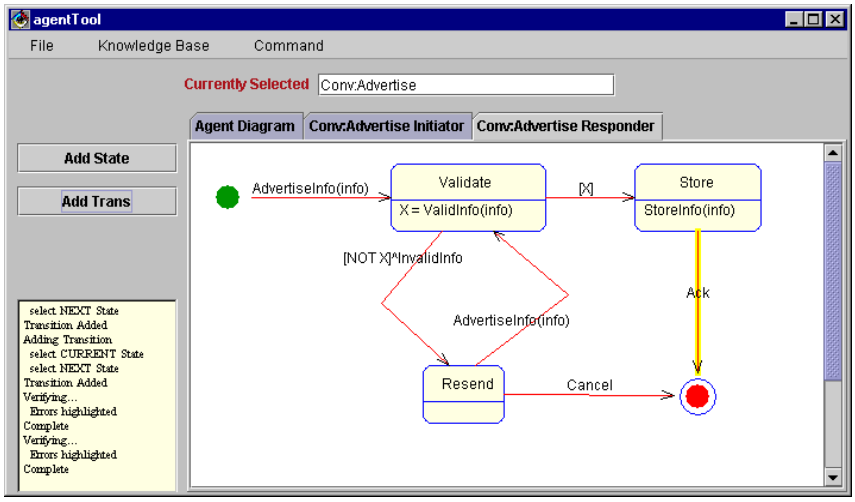

Fig. 10. agentTool Conversation Error

3.3 Assembling Agent Class Components in agentTool

Agent classes in agentTool have internal components that can be added, removed, and manipulated in a manner similar to the other panels of agentTool. Agent classes do have an added layer of complexity however, since all of their components can have Component State Diagrams associated with them and additional sub-components beneath them. The agent class components shown in Figure 11 are the details of the "User" agent class from Figure 9.

Details can also be added to lower levels of abstraction. In Figure 11, the *Component Stat Diag* and *MessageInterface Architecture* tabs lead to a Component State Diagram and Sub-Architecture Diagram respectively. The Component State Diagram defines the dynamic behavior of the component while the Sub-Architecture Diagram contains additional components and connector that further define the component.

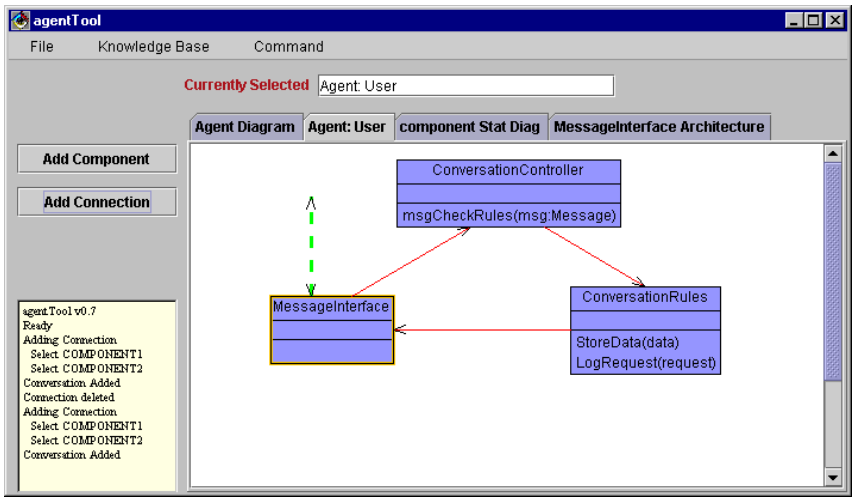

Fig. 11. agentTool Agent Class Components

4 Underlying agentTool Formalisms

Although graphical in nature, the models used in MaSE have a well-defined formal semantics, although space limitations prohibit their introduction here. A more traditional approach toward a formal agent modeling language, CASL, can also be found in this volume [16]. The formal semantics of MaSE are reflected in the transformations from one abstraction to the next. For example, agents *play* roles that *capture* goals and conversations have exactly two participants. These semantics are both incorporated and enforced by agentTool. In a future version of agentTool that incorporates the entire MaSE methodology, a role could be mapped "backward" to the set of goals from which it was created or "forward" to the agent class that plays it.

The agentTool system is based on an object hierarchy that mimics the objects in MaSE. The current agentTool object model is shown in Figure 12. In agentTool, each *System* is composed of a set of *Agents* and *Conversations*. As described above each Agent may have an *Architecture*, which is composed of *Components* and *Connectors*. Likewise, a Conversation is composed of two *State Tables*, which consist of a set of *States* and *Transitions*. Since the internal object model of agentTool only allows the configurations permitted by MaSE, we claim that it formally enforces the MaSE diagram structure and interrelationships.

Since we do not currently have MaSE entirely implemented in agentTool, it is difficult to see how the diagrams from different parts of the methodology are tied together. In our current work, we are extending the agentTool object model to incorporate Goals, Roles, and Tasks. Figure 13 shows this extension to the object model. In the new object model an *Agent* plays at least one *Role*, which consists of one or more *Tasks*. Likewise, all *Roles* must be played by at least one *Agent*. Each Role also captures one or more *Goals* and each *Goal* is captured by exactly one *Role*. This object model makes it easy to see that if the user selects a particular agent, it is

not difficult to determine exactly what *Roles*, *Tasks*, *Goals*, and *Conversations* may be affected by any changes to that *Agent*. What may be even more important is that the user can select a *Goal* and easily determine what *Roles*, *Tasks*, *Agents*, and *Conversations* might be affected by changes to the goal.

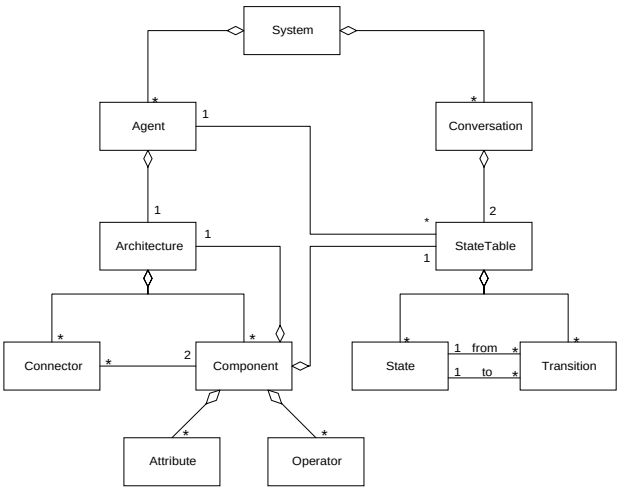

Fig. 12. Current MaSE Object Model

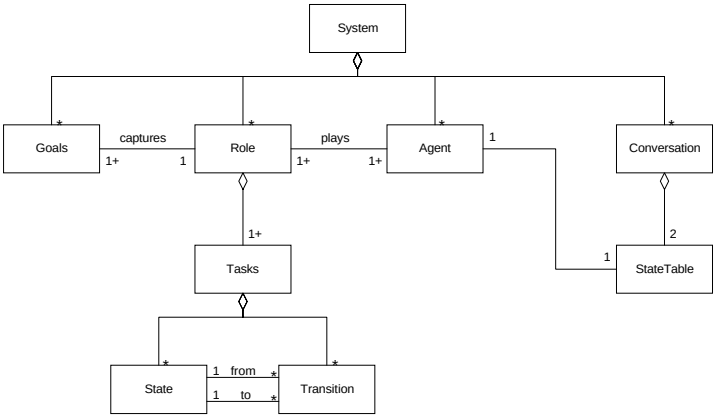

Fig. 13. Extended MaSE Object Model

5 Related Work

There have been several proposed methodologies for analyzing, designing, and building multiagent systems [5]. The majority of these are based on existing object-oriented or knowledge-based methodologies. The most widely published methodology is Gaia methodology [17], which we will use for comparison.

While Gaia and MaSE take a similar approach in analyzing multiagent system, MaSE is more detailed and provides more guidance to system designer. The first step in Gaia is to extract roles from the problem specification without real guidance on how this is done. MaSE, on the other hand, develops Goal Hierarchy and Use Cases to help define what roles should be developed. The use of roles by both methodologies is similar. They are used to abstractly define the basic behaviors within the system. In Gaia, roles define functionality in terms *responsibilities*. In MaSE, similar functionality is defined in Concurrent Task Diagram; however, concurrent tasks provide communication detail not found in Gaia. Gaia *permissions* define the use of resources by a particular role, which is not yet defined in MaSE. Gaia uses *activities* to model computations performed within roles and *protocols* to define interactions with other roles. These are captured with MaSE tasks, which once again provide more detail about when actions are performed, data input and output, and how the actions relate to the interaction protocols.

In the design phase, Gaia only provides a high-level design. It consists of an Agent Model, which identifies the agent types (and the roles from which they were derived) and their multiplicity within the system. This information is captured in MaSE Agent Class Diagrams and Deployment Diagrams. The Gaia Services Model identifies the main services of the agent type in terms of the inputs, outputs, and pre- and post-conditions. This does not have a direct parallel in MaSE although services and the details of the interactions (inputs and outputs) are defined in MaSE tasks and conversations. Finally, the Gaia acquaintance model identifies lines of communications between agent types. In MaSE, this information is captured in the Agent Class Diagram, which also identifies the types of interactions as individual conversations.

6 Conclusions and Future Work

The Multiagent Systems Engineering methodology is a seven-step process that guides a designer in transforming a set of requirements into a successively more concrete sequence of models. By analyzing the system as a set of roles and tasks, a system designer is naturally lead to the definition of autonomous, pro-active agents that coordinate their actions to solve the overall system goals.

MaSE begins in the analysis phase by capturing the essence of an initial system context in a structured set of goals. Next, a set of use cases are captured and transformed into Sequence Diagrams so desired event sequences will be designed into the system. Finally, the goals are combined to form roles, which include tasks that describe how roles satisfy their associated goals. In the design phase, roles are combined to define agent classes and tasks are used to identify conversations between the classes. To complete the agent design, the internal agent architecture is chosen and actions are mapped to functions in the architecture. Finally, the run-time structure of the system is defined in a Deployment Diagram and implementation choices such as language and communication framework are made.

MaSE, and our current version of agentTool, has been used to develop five to ten small to medium sized multiagent systems ranging from information systems [8, 12]

and mixed-initiative distributed planners [2] to biologically based immune systems [4]. The results have been promising. Users tell us that following MaSE is relatively simple, yet is flexible enough to allow for a variety of solutions. We are currently using MaSE and agentTool to develop larger scale multiagent systems that are both mobile and dynamic in nature.

From our research on MaSE and agentTool, we have learned many lessons. First, it is clear that developing a methodology with an eye towards automation and formally defined relationships between the various models simplifies the semantics and makes implementation much easier. Secondly, using object-oriented principles as a basis for our methodology was the right choice. We consider MaSE a domain specific instance of the more general object-oriented paradigm. This also simplifies the underlying formal model as well as code generation. Instead of dealing with a general association, we have just one – a conversation between agents. While agents are not equivalent to an object, they are a specialization. Once again, we can focus our methodology and tool thus making the entire process less complex. Finally, we have shown that you can develop a methodology and tool to support multiple types of agent architectures, languages, and communications frameworks.

As stated throughout this paper, MaSE and agentTool are works in progress. We are currently extending MaSE to handle mobility and dynamic systems (in terms of agents being able to enter and leave the system during execution). We are also looking more closely at the relationship between tasks, conversations, and the internal design of agents. As for agentTool, we are extending it to handle all aspects of MaSE including code generation. We currently have a code generator that generates complete conversations for a single communication framework.

Acknowledgements

This research was supported by the Air Force Office of Scientific Research (AFOSR) and the Dayton Area Graduate Studies Institute (DAGSI). 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.

References

1. F. Bellifemine, A. Poggi, and G. Rimassa. Developing Multi-Agent Systems with JADE. In C. Castelfranchi and Y. Lespérance, editors, *Intelligent Agents VII. Agent Theories, Architectures, and Languages - 7th. International Workshop, ATAL-2000, Boston, MA, USA, July 7-9, 2000, Proceedings*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, 2001. In this volume.
2. M. Cox, B. Kerkez, C. Srinivas, G. Edwin, and W. Archer. Toward Agent-Based Mixed-Initiative Interfaces. *Proceedings of the International Conference on Artificial Intelligence (IC-AI) 2000*. pages 309-316. CSREA Press, 2000.
3. S. DeLoach. Multiagent Systems Engineering: a Methodology and Language for Designing Agent Systems. *Proceedings of Agent Oriented Information Systems '99*. pages 45-57, 1999.

4. P. Harmer and G. Lamont. An Agent Architecture for a Computer Virus Immune System. *Workshop on Artificial Immune Systems at Genetic and Evolutionary Computation Conference*, Las Vegas, Nevada, July 2000.
5. C. Iglesias, M. Garijo, and J. Gonzalez. A Survey of Agent-Oriented Methodologies. In: Müller, J.P., Singh, M.P., Rao, A.S., (Eds.): *Intelligent Agents V. Agents Theories, Architectures, and Languages*. Lecture Notes in Computer Science, Vol. 1555. Springer-Verlag, Berlin Heidelberg, 1998.
6. E. Kendall. Agent Roles and Role Models: New Abstractions for Multiagent System Analysis and Design. *Proceedings of the International Workshop on Intelligent Agents in Information and Process Management*, Bremen, Germany, September 1998.
7. E. Kendall, U. Palanivelan, and S. Kalikivayi. Capturing and Structuring Goals: Analysis Patterns. *Proceedings of the Third European Conference on Pattern Languages of Programming and Computing*, Bad Irsee, Germany, July 1998.
8. S. Kern, M. Cox, and M. Talbert. A Problem Representation Approach for Decision Support Systems. *Proceedings of the Eleventh Annual Midwest Artificial Intelligence and Cognitive Science Conference*, pages 68-73. AAAI, 2000.
9. D. Kinny, M. Georgeff, and A. Rao. A Methodology and Modelling Technique for Systems of BDI Agents. *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW '96.* LNAI volume 1038, pages 56-71, Springer-Verlag, 1996.
10. T. Lacey, S. DeLoach. Automatic Verification of Multiagent Conversations. *Proceedings of the Eleventh Annual Midwest Artificial Intelligence and Cognitive Science Conference*, pages 93-100. AAAI, 2000.
11. S. Marini, M. Martelli, V. Mascardi, and F. Zini. Specification of heterogeneous agent architectures. In C. Castelfranchi and Y. Lespérance, editors, *Intelligent Agents VII. Agent Theories, Architectures, and Languages - 7th. International Workshop, ATAL-2000, Boston, MA, USA, July 7-9, 2000, Proceedings*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, 2001. In this volume.
12. J. McDonald, M. Talbert, and S. DeLoach. Heterogeneous Database Integration Using Agent Oriented Information Systems. *Proceedings of the International Conference on Artificial Intelligence (IC-AI) 2000*. pages 1359-1366, CSREA Press, 2000.
13. H. Nwana, D. Ndumu Leel, and J. Collis. ZEUS: A Toolkit for Building Distributed Multi-Agent Systems. *Applied Artificial Intelligence Journal*. **13** (1), pages 129-185, 1999.
14. M. Raphael. *Knowledge Base Support for Design and Synthesis of Multi-agent Systems.* MS thesis, AFIT/ENG/00M-21. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base Ohio, USA, March 2000.
15. D. Robinson. *A Component Based Approach to Agent Specification*. MS thesis, AFIT/ENG/00M-22. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base Ohio, USA, March 2000.
16. S. Shapiro and Y. Lespérance. Modeling multiagent systems with CASL - a feature interaction resolution application. In C. Castelfranchi and Y. Lespérance, editors, *Intelligent Agents VII. Agent Theories, Architectures, and Languages - 7th. International Workshop, ATAL-2000, Boston, MA, USA, July 7-9, 2000, Proceedings*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, 2001. In this volume.
17. M. Wooldridge, N. Jennings, and D. Kinny. The Gaia Methodology for Agent-Oriented Analysis and Design. *Journal of Autonomous Agents and Multi-Agent Systems*, 3(3), 2000.