presented at 12th Midwest Artificial Intelligence and Cognitive Science Conference (MAICS 2001)
Miami University, Oxford, Ohio, March 31 - April 1, 2001

Analysis and Design using MaSE and agentTool

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
sdeloach@computer.org

Abstract

This paper provides an overview of the work being done at the Air Force Institute of Technology on the Multiagent Systems Engineering methodology and the associated agentTool environment. Our research is focused on discovering methods and techniques for engineering practical multiagent systems. It uses the abstraction provided by multiagent systems for developing intelligent, distributed software systems.

Introduction

Multiagent systems have brought together many disciplines in an effort to build distributed, intelligent, and robust 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 (Iglesias, Garijo & Gonzalez 1998). Unfortunately, most of the tools and methodologies have either focused on specific agent architectures or lacked sufficient detail to adequately support designing complex systems. In our research, we have developed both a complete-lifecycle methodology and a complimentary environment for analyzing, designing, and developing heterogeneous multiagent systems. The methodology we are developing is Multiagent Systems Engineering (MaSE) (DeLoach, Wood, & Sparkman 2000) while the tool we are building to support that methodology is agentTool (Wood & DeLoach 2001).

In our research, we do not take the typical artificial intelligence view of agents where agents are required to be autonomous, proactive, reactive, and social. For our purposes, agents are simple software processes that interact with each other to meet an overall system goal. It is often the case that multiple, non-complex agents may interact in such as way that the entire system may exhibit seemingly intelligent behavior. 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. Our work is aimed at the much larger problem of building complex, distributed, and possibly dynamic systems that will pervade the future of computing. To build these complex systems, distributed agents must work cooperatively with other agents in a heterogeneous environment.

Sycara (Sycara 1998) describes six challenges of multiagent systems as

1. decomposing problems and allocating tasks to individual agents,
2. coordinating agent control and communications,
3. making multiple agents act in a coherent manner,
4. reasoning about other agents and the state of coordination,
5. reconciling conflicting goals between agents, and
6. engineering practical multiagent systems.

Our research 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. To accomplish this goal, MaSE uses a number of graphically based models to describe the types of agents in a system and their interfaces to other agents, as well as an architecture-independent definition of the internal agent design.

Multiagent Systems Engineering

In general, our research at AFIT has focused on developing the methodology, techniques, and tools for building practical agent systems. To this end, we have developed the Multiagent Systems Engineering methodology (Wood & DeLoach 2001, DeLoach & Wood 2001) that defines multiagent systems in terms of agent classes and their organization. We define their organization in terms of which agents can communicate using conversations. There are two basic phases in MaSE: analysis and design. The first phase, Analysis, includes three steps: capturing goals, applying use cases, and refining roles.

The first step, Capturing Goals, takes user requirements and turns them and top-level system goals. After defining system level goals, we extract system-level use cases and define Sequence Charts in the applying use cases step. This step defines an initial set of system roles and communications paths. Using the system goals and roles

identified in the use cases, we refine and extend the initial set of roles and define tasks to accomplish each goal in the refining roles step.

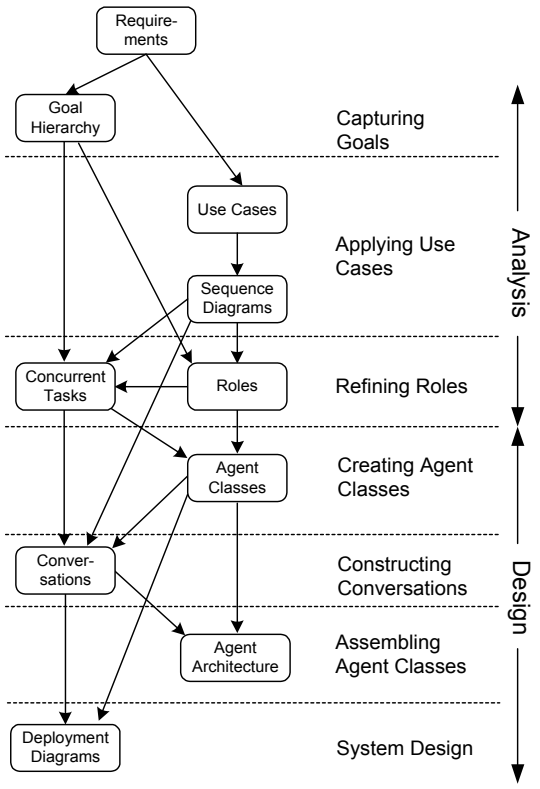

Figure 1. MaSE Methodology

In the Design phase, we transform the analysis models into constructs useful for actually implementing the multiagent system. The Design phase has four steps: creating agent classes, constructing conversations, assembling agent classes, and system design. In the first step, creating agent classes, we define specific agent classes to fill the roles defined in the Analysis phase. Then, after determining the number and types of agent classes in the system, we can either construct conversations between those agent classes or define the internal components that comprise the agent classes. The analyst may perform these steps in parallel during the constructing conversations and assembling agent classes steps. Once we have completely defined the system structure, we define how the system is to be deployed. During this step, the designer defines the number of individual agents, their locations, and other system specific items.

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. A MaSE *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.

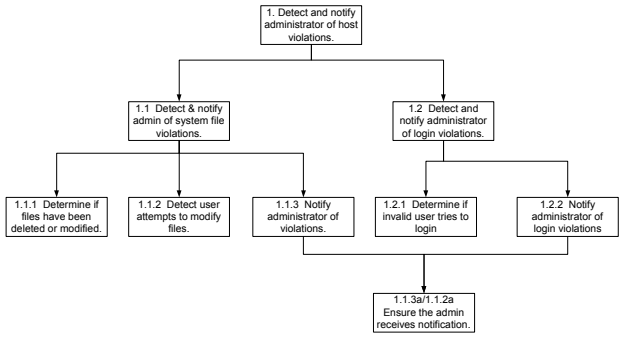

Figure 2. Goal Hierarchy Diagram

There are two steps to Capturing Goals: identifying the goals and structuring goals. An analyst may identify goals by distilling the essence of the set of requirements. These requirements may include detailed technical documents, user stories, or formalized specifications. Once captured and explicitly stated, goals are less likely to change than the detailed steps and activities involved in accomplishing them (Kendall, Palanivelan & Klikivayi 1998). Next, the analyst analyses and structures the identified goals into a Goal Hierarchy Diagram. In a Goal Hierarchy Diagram, goals the analyst organizes the goals by importance. Each level of the hierarchy should contain goals that are roughly equal in scope. The analyst also identifies sub-goals that are necessary to satisfy parent goals. Eventually, the analyst will associate each goal with a role and a set of agent classes responsible for satisfying that goal.

Applying Use Cases

The Applying Uses Cases step is a crucial step in translating goals into roles and associated tasks. The analyst draws use cases from the system requirements and users. *Use cases* 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 analyst restructures the use cases into 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. In the next step, the analyst will use the events identified here to help define tasks and, eventually, conversations.

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

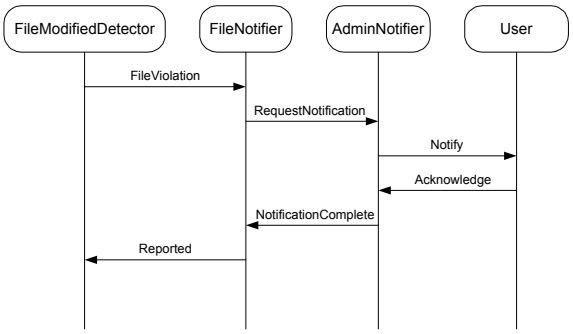

Figure 3. Sequence Diagram

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, which 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 (Kendall 1998). Each goal is generally mapped to a single role. However, there are 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.

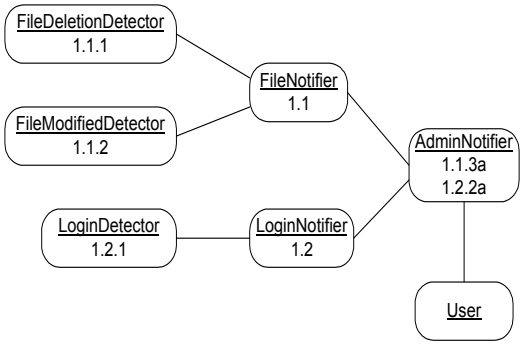

Figure 4. Role Model

Once roles have been defined, tasks are created. The most interesting, and most difficult, part of using the MaSE methodology is transforming the roles into agent classes and defining the conversations and internal agent behaviors. To help us accomplish this task, we need to be able to define high-level tasks that can be transformed into specific agent functionality. This functionality helps us define the internal components of agents as well as the details of the conversations in which the agents participate. Figure 5 shows a detailed version of the MaSE role model. The ovals below each role denote tasks that the role must execute in order to accomplish its goal. The lines between nodes indicate protocols between tasks. These protocols define a series of messages between the tasks that allow them to work cooperatively. The arrows on the protocol lines point from the initiator of the protocol to the responding task.

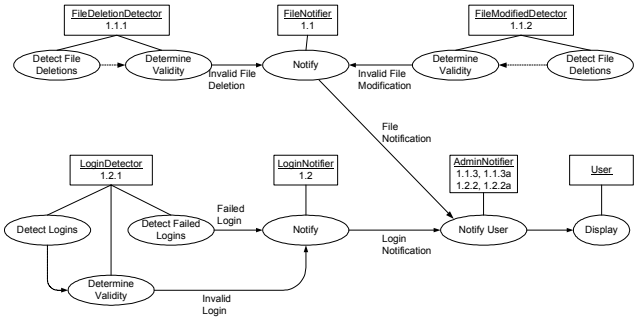

Figure 5. MaSE Role Model

We define these concurrent tasks (DeLoach 2001) as a finite state automaton that specifies messages sent between roles and tasks. Concurrent tasks also allow us to specify internal processing via activities in the states. Using concurrent tasks, we can define higher level, complex interaction protocols that require coordination between multiple agents. We have also shown that we can actually verify correct operation of such interaction protocols based on Concurrent Tasks (Lacey & DeLoach 2000a). An example of a MaSE Concurrent Task Diagram, which defines the Notify User task of the AdminNotifier role, is shown in Figure 6.

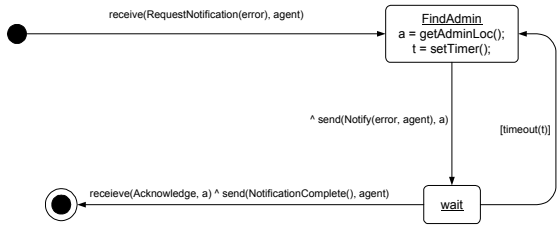

Figure 6. Concurrent Task Diagram

The syntax of a transition follows the notation below.

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

We interpret the transition to say that if an event trigger is received with a number of arguments *args1* 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.

Creating Agent Classes

In Creating Agent Classes, agent classes are identified from roles and documented in an Agent Class Diagram, as

shown in Figure 7. 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.

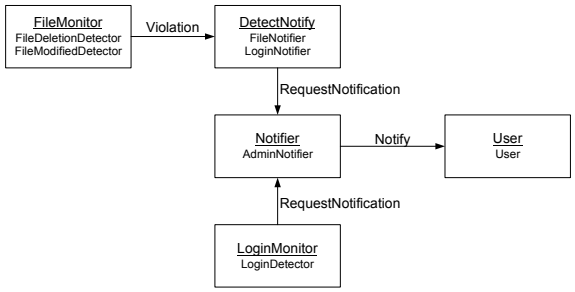

Figure 7. Agent Class Diagram

Constructing Conversations

The designer may perform the next two steps, Constructing Conversations and Assembling Agents in parallel. The two steps are closely linked, since 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 8. 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.

Assembling Agents

In this step of MaSE, the internals of agent classes are created. Robinson (Robinson 2000) 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.

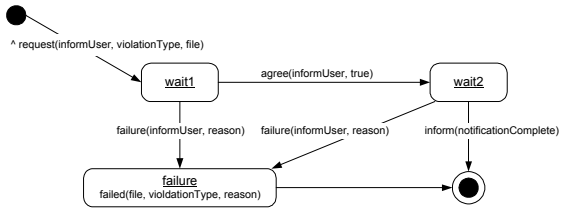

Figure 8. Communication Class Diagram

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.

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.

agentTool

The agentTool system is our attempt to implement a tool to support and enforce MaSE. Currently agentTool implements all seven steps of MaSE as well as automated support for transforming analysis models into design models. 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 (Raphael & DeLoach 2000) conversation verification (Lacey & DeLoach 2000a) 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 10, three roles have been defined with their various collections of concurrent tasks. When the user selects the Register Researcher task (by clicking on the oval), the Task Panel

tab becomes visible. The user may then access that diagram (Figure 11) by selecting the appropriate tab.

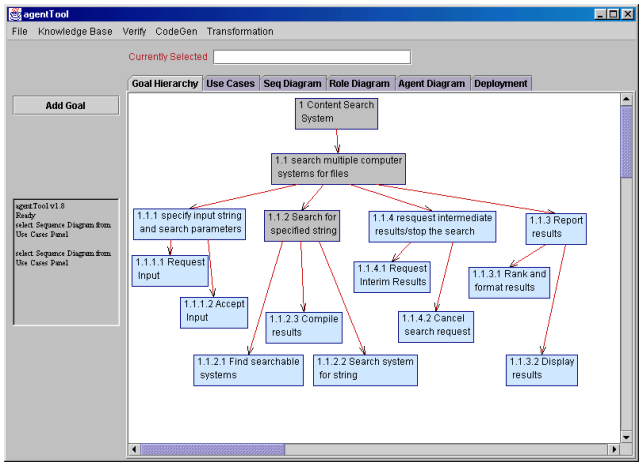

Figure 9. agentTool Goal Hierarchy Diagram

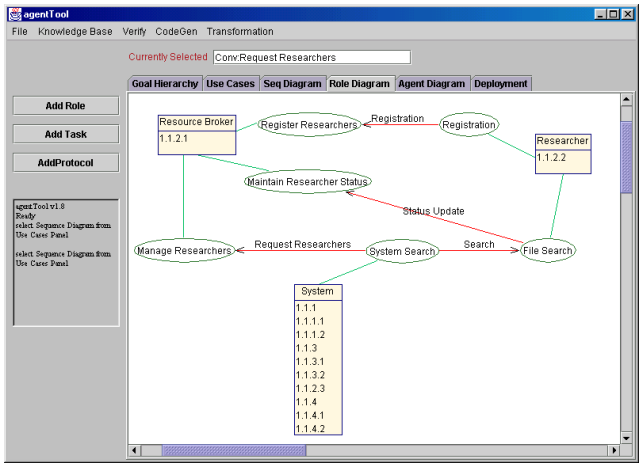

Figure 10. agentTool Role Model

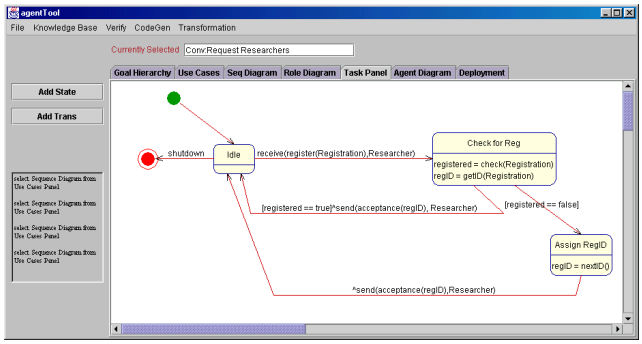

Figure 11. agentTool Concurrent Task Diagram

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.

During each step of system development, the various analysis and design diagrams are 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 currently selected object controls the available diagrams (via tabs), which include those that can be reached following valid MaSE steps. For instance, selecting a task causes a tab for the associated Concurrent Task Diagram to become visible. Selecting that tab would cause the Concurrent Task Diagram to appear.

Designing Systems using agentTool

Designing a multiagent system using agentTool begins in an Agent Class Diagram as shown above in Figure 12. Since a conversation can only exist between agent classes, we must define agent classes before we can define the 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. AgentTool supports either method, which is generally a matter of personal choice.

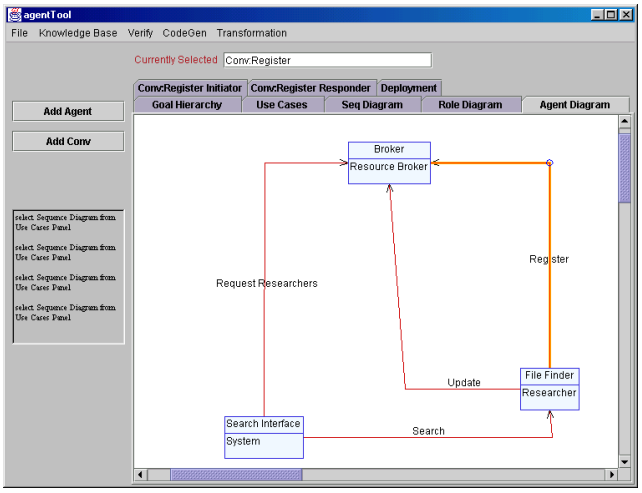

Figure 12. agentTool Class Diagram

Once we have defined agent classes and conversations, we can define the details of the conversations using Communication Class Diagrams (Figure 13). The "Add State" button adds a state to the panel while the "Add Trans" button adds a conversation between the two selected states. A designer can verify a conversations at any point during its creation by using the Verify Conversations command from the Verify menu (Lacey & DeLoach 2000b). The agentTool verification process ensures

conversation specifications are deadlock free. If any errors exist, the verification results in a highlighted section of a conversation, as shown in Figure 13 on the "Ack" transition (highlights are yellow in the application). Each highlight indicates a potential error as detected by the verification routine.

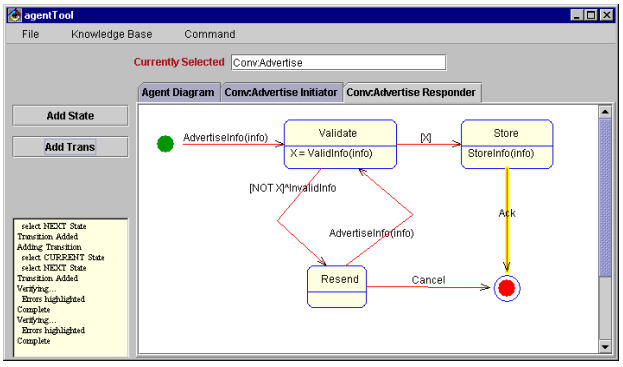

Figure 13. agentTool Conversation Error

Agent classes have internal components that can be added, removed, and manipulated in a manner similar to the other agentTool panels (see Figure 14). However, agent classes do have an added layer of complexity since components can have internal Component State Diagrams and possibly additional sub-components beneath them.

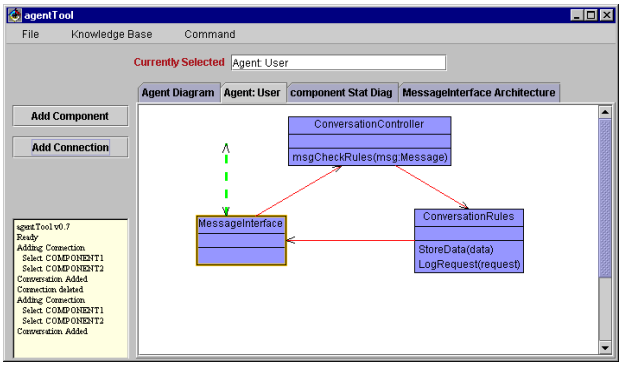

Figure 14. agentTool Agent Class Components

The designer can add detailed design information at a lower levels of abstraction. In Figure 14, the Component State Diagram 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.

Semi-automated Design Support

Recent work on agentTool includes developing support for semi-automatic transformations that transform a set of analysis models into the appropriate design models (Sparkman 2001). To perform the process, the designer must first assign roles to specific agent classes. After the assignments have been made, the designer can apply the semi-automated transformations to the analysis models. There are basically three stages to the transformations. In stage one, the transformations attempt to determine to which protocol events in the concurrent tasks belong. In most cases this can be done automatically. However, in some cases, the system cannot precisely determine the appropriate protocol for each send/receive event. When this happens, the system simply asks the designer to make the choice as shown in Figure 15. Here the system could not automatically determine to which protocol the bottom *receive(msg, ag2)* event actually belongs. In this case, the designer has the option to choose *Protocol2*, *Protocol3*, or both. Once the designer has made the choice, the system carries out the remaining transformations.

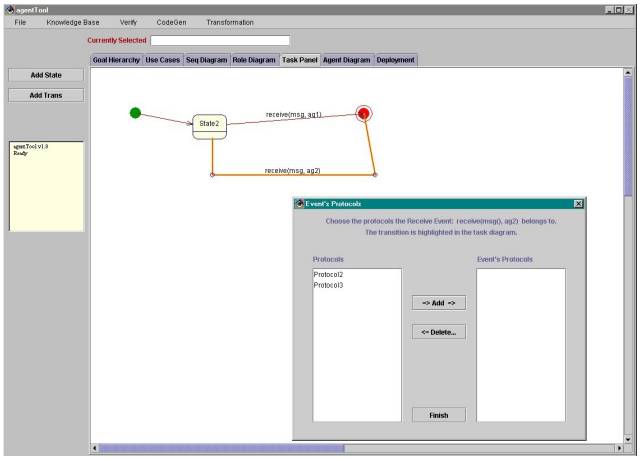

Figure 15. Semi-automatic Transformations - Getting User Input

After the protocols have been determined for each event, the transformation continues by creating internal agent components for each concurrent task associated with the roles being played by an agent. Then, the system copies the concurrent task definition into the component's internal state machine. This ensures the behavior defined for each role is transferred to the agent that will play that role.

In stage two, the internal state machines of each component are annotated in preparation for extracting the actual conversations. This preparation phase finds the starting and ending location of each conversation and ensures that conversations between agents "match up."

In the final stage, the conversations are extracted from the internal components and placed in separate conversation diagrams. The conversations are replaced by method calls so that the internal component state machine still retain internal processing and allow for conversation coordination.

Mobility

Other recent work on MaSE and agentTool includes research into providing the ability to model mobile agents.

The initial step in modeling mobility is to include a move activity in a concurrent task state. The *move* activity basically requests that the agent move to a new location. The actual implementation of the move activity is assumed to be part of the environment within which the agent is executing. The activity returns a Boolean value as to whether the move actually occurred. This simple addition to the analysis phase allows the analyst to specify when a move should occur, the requested location, and the ability to decide whether or not the move was successful.

While simple to model in the analysis phase, mobility is more complex in the design phase. At the design level, MaSE had to provide the capability to inform each component when a move was requested and to provide the capability for each component to store its current state, shutdown, and restart after the move. To help the designer carry out these complex design activities, semi-automated transformations, similar to those described in the previous section, were developed and implemented in agentTool (Self 2001).

Summary

This paper has presented an overview of the Multiagent Systems Engineering methodology and the agentTool environment. MaSE and agentTool are being developed in tandem to provide guidance and practical support for building complex, distributed, and dynamic systems. This research is supported by grants from the Air Force Office of Scientific Research and the Dayton Area Graduate Studies Institute. 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. S. DeLoach, M. Wood, and C. Sparkman. Multiagent Systems Engineering, submitted to *International Journal of Software Engineering and Knowledge Engineering*, November 2000.
2. S. DeLoach, Specifying Agent Behavior as Concurrent Tasks, *to appear at the 5th International Conference on Autonomous Agents*, Montreal, Canada. May 28 – June 1, 2001.
3. Scott A. DeLoach & Mark Wood. Developing Multiagent Systems with agentTool, *Intelligent Agents VII - Proceedings of the 7th International Workshop on Agent Theories, Architectures, and Languages (ATAL'2000)*. Springer Lecture Notes in AI, Springer Verlag, Berlin, 2001.
4. 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.
5. 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.
6. 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.
7. T. Lacey, S. DeLoach, Verification of Agent Behavioral Models. *Proceedings of the International Conference on Artificial Intelligence* (IC-AI'2000), pages 557-564, CSREA Press, 2000.
8. 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.
9. Marc J. Raphael & Scott A. DeLoach. A Knowledge Base for Knowledge-Based Multiagent System Construction, *National Aerospace and Electronics Conference (NAECON)*, Dayton, OH, October 10-12, 2000.
10. 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.
11. A. Self. *Design & Specification of Dynamic, Mobile, and Reconfigurable Multiagent Systems*. MS thesis, AFIT/ENG/01M-11. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base Ohio, USA, March 2001.
12. C. Sparkman. *Transforming Analysis Models into Design Models for the Multiagent Systems Engineering Methodology*. MS thesis, AFIT/ENG/01M-12. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base Ohio, USA, March 2001.
13. K. P. Sycara, "Multiagent Systems," *AI Magazine* vol. 19(2), pp. 79-92, 1998.
14. M. Wood, S. DeLoach. An Overview of the Multiagent Systems Engineering Methodology, In P. Ciancarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering - First International Workshop (AOSE), Limerick, Ireland, June 10, 2000.* Lecture Notes in Computer Science. Vol. 1957, Springer Verlag, Berlin, 2001.