Chapter 6

THE MASE METHODOLOGY

Scott A. DeLoach

Abstract MaSE provides a detailed approach to the analysis and design of MAS. MaSE combines several established models into a comprehensive methodology and provides a set of transformation steps that shows how to derive new models from the existing models. Thus MaSE guides the developer in the analysis and design process. Future work on MaSE will focus on specializing it for use in adaptive multiagent and cooperative robotic systems based on an organizational theoretic approach. We are currently developing an organizational model that will provide the knowledge required for a team of software or hardware agents to automatically adapt to changes in their environment and to organize and re-organize to accomplish team goals. Much of the information needed in this organizational model – goals, roles, and agents – is already captured in MaSE. However, we will have to extend MaSE analysis to capture more detail on roles, including the capabilities required to play roles.

1. Introduction

This chapter provides an introduction to Multiagent Systems Engineering (MaSE), which is a full-lifecycle methodology for analyzing, designing, and developing heterogeneous MAS. To accomplish this, MaSE uses a number of graphically based models derived from standard UML models to describe the types of agents in a system and their interfaces to other agents, as well as an architecture-independent detailed definition of the internal agent design. The primary focus of MaSE is to guide a designer from an initial set of requirements through the analysis, design, and implementation of a working MAS.

MaSE views MAS as a further abstraction of the object-oriented paradigm where agents are specialized objects. Instead of simple objects, with methods that can be invoked by other objects, agents coordinate with each other via conversations and act proactively to accomplish individual and system-wide goals. Therefore, MaSE builds upon well-founded object-oriented techniques and applies them to the specification and design of MAS.

MaSE is also the basis for the *agentTool* development system. agentTool can be downloaded free from the agentTool Web page at `http://www.cis.ksu.edu/~sdeloach`. agentTool is a graphically based, fully interactive software engineering tool, which fully supports each step of MaSE analysis and design. agentTool also supports automatic verification of inter-agent communications, semi-automated design, and code generation for multiple MAS frameworks. MaSE and agentTool are both independent of any particular agent architecture, programming language, or communication framework.

2. Methodology

The MaSE methodology is a specialization of more traditional software engineering methodologies. The general operation of MaSE follows the phases and steps shown below and uses the associated models.

Phases	**Models**
1. Analysis Phase	
a. Capturing Goals	Goal Hierarchy
b. Applying Use Cases	Use Cases, Sequence Diagrams
c. Refining Roles	Concurrent Tasks, Role Model
2. Design Phase	
a. Creating Agent Classes	Agent Class Diagrams
b. Constructing Conversations	Conversation Diagrams
c. Assembling Agent Classes	Agent Architecture Diagrams
d. System Design	Deployment Diagrams

The MaSE Analysis phase consists of three steps: Capturing Goals, Applying Use Cases, and Refining Roles. The Design phase has four steps: Creating Agent Classes, Constructing Conversations, Assembling Agent Classes, and System Design. While presented sequentially, the methodology is, in practice, iterative. The intent is that the designer is free to move between steps and phases such that with each successive pass, additional detail is added and, eventually, a complete and consistent system design is produced.

One strength of MaSE is the ability to track changes during the whole process. Every object created during the analysis and design phases can be traced forward or backward through the different steps to other related objects. For instance, a goal derived in the Capturing Goals step can be traced to a specific role, task, and agent class. Likewise, an agent class can be traced back through tasks and roles to the system level goal it was designed to satisfy.

3. Analysis Phase

The MaSE Analysis phase produces a set of roles and tasks, which describe how a system satisfies its overall goals. Goals are an abstraction of the detailed

requirements and are achieved by roles. Typically, a system has an overall goal and a set of sub-goals that must be achieved to reach the system goal. Goals are used in MaSE because they capture *what* the system is trying to achieve and tend to be more stable over time than functions, processes, or information structures.

A *role* describes an entity that performs some function within the system. In MaSE, each role is responsible for achieving, or helping to achieve specific system goals or sub-goals. MaSE roles are analogous to roles played by actors in a play or by members of a typical company structure. The company (which corresponds to system) has roles such as "president," "vice-president," and "mail clerk" that have specific responsibilities, rights and relationships defined in order to meet the overall company goal.

The overall approach in the MaSE Analysis phase is straightforward: define system goals from a set of requirements and then define the roles necessary to meet those goals. To help in defining roles to meet specific goals, MaSE uses Use Cases and Sequence Diagrams. The individual steps of the Analysis phase of Capturing Goals, Applying Use Cases, and Refining Roles are presented next.

3.1 Capturing Goals

The first step in the MaSE Analysis phase is Capturing Goals, whose purpose is to transform an initial system specification into set of structured system goals. The *initial system context,* the starting point for MaSE analysis, is usually a software requirement specification with a well-defined set of requirements. These requirements tell the analyst the services that the system must provide and how the system should or should not behave based on inputs to the system and its current state. There are two sub-steps in Capturing Goals: identifying goals and structuring goals. First, goals must be identified from the initial system context. Next, the goals are analyzed and put into a hierarchical form. Each of these sub-steps is described in detail below.

Identifying Goals. The goal of the step named Identifying Goals is to capture the essence of an initial set of requirements. This process begins by extracting scenarios from the initial specification and describing the goal of that scenario.

Throughout this chapter, we will use the conference management system example, which has become fairly common in AOSE circles. The conference management system is a MAS supporting the management of various sized international conferences that require the coordination of several individuals and groups. We define the basic system requirements below.

- Authors should be able to submit their papers electronically to a conference paper database system. During the submission phase, authors should be notified of paper receipt and given a paper submission number.
- After the deadline for submissions has passed, the papers will be divided among the program committee (PC), who has to review the papers themselves or by contacting referees and asking them to review a number of the papers.
- Reviewers should be able to get papers directly from the central database and submit their reviews to a central collection point.
- After the reviews are complete, a decision on accepting or rejecting each paper must be made.
- After the decisions are made, authors are notified of the decisions and are asked to produce a final version of their paper if it was accepted.

The conference management system is an organization whose membership (authors, reviewers, decision makers, review collectors, etc.) may change during the process. In addition, since each agent is associated with a human, it is easy to imagine that these agents could be coerced into displaying opportunistic behaviors that would benefit their owner to the detriment to the overall system. Such behaviors could include reviewing ones own paper or inequitable allocation of work, etc.

An example of the goals derived from these requirements is shown below. Notice that all the details on how to perform system functions are not included as goals.

1. Collect papers
2. Distribute papers
3. Assign papers to PC members
4. Assign papers to reviewers
5. Submit reviews
6. Collect reviews
7. Select/reject papers
8. Inform authors

Goals embody the critical system requirements; therefore, an analyst should specify goals as abstractly as possible without losing the spirit of the requirement. This abstraction can be performed by removing detailed information when specifying goals. For example, to "Inform authors" is a goal, the details on how to actually inform them may change over time and are not.

Once the goals have been captured, they provide the foundation for the analysis model; all roles and tasks defined in later steps must support one of the goals. If, later in the analysis, the analyst discovers roles or tasks that do not support an existing system goal, either the roles and tasks are superfluous or a new goal has been discovered.

Structuring Goals. The final step in Capturing Goals is structuring the goals into a Goal Hierarchy Diagram, as shown in Figure 6.1. A Goal Hierarchy Diagram is a directed, acyclic graph where the nodes represent goals and

the arcs define a sub-goal relationship. A goal hierarchy is not necessarily a tree as a goal may be a sub-goal of more than one parent goal.

To develop the goal hierarchy, the analyst studies the goals for their importance and inter-relationships. Even though goals have been captured, they are of various importance, size, and level of detail. The Goal Hierarchy Diagram preserves such relationships, and divides goals into sub-goals that are easier to manage and understand.

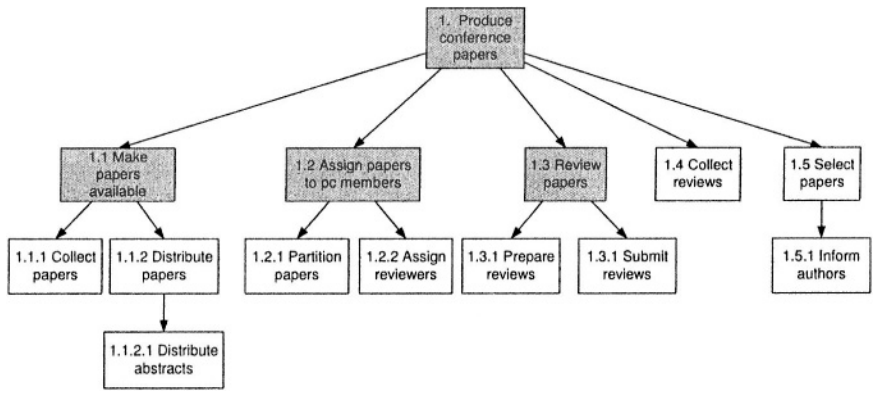

Figure 6.1. Example Goal Hierarchy Diagram

The first step in building is to identify the overall system goal, which is placed at the top of the Goal Hierarchy Diagram. However, it is often the case, as in our example above, that a single system goal cannot be directly extracted from the basic requirements. In this case, the highest-level goals are summarized to create an overall system, in our case "Produce conference papers." Once a basic goal hierarchy is in place, goals may be decomposed into new sub-goals. Each sub-goal must support its parent goal in the hierarchy and defines *what* must done to accomplish the parent goal.

Although similar, Goal decomposition is not simply "functional decomposition." Goals describe *what,* while functions describe *how.* Instead of a set of goals describing *what* the system will do, functional decomposition typically results in a set of steps prescribing *how* the system will do it. For example, functional steps for implementing the goal "Assign papers to PC members" might be to *(i)* group papers based on similar keywords; and *(ii)* select PC members whose expertise matches the paper groups. However, the appropriate sub-goals would be to: *(i)* "Partition papers"; and *(ii)* "Assign reviewers." The fact that the papers are partitioned and PC members are assigned to papers are goals, *how* we divide the papers or on what basis we assign reviewers are immaterial at this point and will be decided on by the agents responsible for those goals. Goal decomposition continues until any further decomposition

would result in functions instead of a goals (i.e., the analyst prescribes *how* a goal should be accomplished).

There are four special types of goals in a Goal Hierarchy Diagram. These are: summary, partitioned, combined, and non-functional. Goals can have attributes of more than one special goal type; however, they do not necessarily have to be one of these types at all.

A *summary goal* is derived from a set of existing "peer" goals to provide a common parent goal. This often happens at the highest levels of the hierarchy as was the case in the overall system goal in our example.

Some goals do not functionally support the overall system goal, but are critical to system operation. These *non-functional goals* are often derived from non-functional requirements such as reliability or response times. For example, if a system must be able to find resources dynamically, a goal to facilitate locating dynamic resources may be required. In this case, another "branch" of the Goal Hierarchy Diagram can be created and placed under an overall system level goal.

There are often a number of sub-goals in a hierarchy that are identical or very similar that can be grouped into a *combined goal.* This often happens when the same basic goal is a sub-goal of two different goals. In this case, the combined goal becomes a sub-goal of both the goals.

A *partitioned goal* is a goal with a set of sub-goals that, when taken collectively, effectively meet that goal. While this is always true of summary goals, it may be true of any goals with a set of sub-goals. By defining a goal as "partitioned," it frees the analyst from specifically accounting for it in the rest of the analysis process. Partitioned goals are annotated in a Goal Hierarchy Diagram using a gray goal box instead of a clear box (e.g., goals 1, 1.1, and 1.2 in Figure 6.1).

At the conclusion of Capturing Goals, system goals have been captured and structured into a Goal Hierarchy Diagram. The analyst can now move to the second Analysis step, Applying Use Cases, where the initial look at roles and communication paths takes place.

3.2 Applying Use Cases

The Applying Uses Cases step is crucial in translating goals into roles and associated tasks. Use cases are drawn from the system requirements and describe sequences of events that define desired system behavior; they are examples of how the system should behave. To help determine the actual communications in a MAS, the use cases are converted into Sequence Diagrams. MaSE Sequence Diagrams are similar to standard UML sequence diagrams except that they are used to depict sequences of events between *roles* and to define the communications between the agents that will be playing those roles. The roles

identified here form the initial set of roles used in the next step while the events are also used later to define tasks and conversations.

The first step in Applying Use Cases is to extract Use Cases from the initial system context, which should include both positive and negative Use Cases. A *positive Use Case* describes what should happen during normal system operation. However, a *negative Use Case* defines a breakdown or error. While Use Cases cannot be used to capture every possible requirement, they are an aid in deriving communication paths and roles. Cross checking the final analysis against the set of derived goals and Use Cases provides a redundant method for deriving system behavior.

3.3 Refining Roles

The purpose of the Refining Roles step is to transform the Goal Hierarchy Diagram and Sequence Diagrams into roles and their associated tasks, which are forms more suitable for designing MAS. Roles form the foundation for agent classes and correspond to system goals during the Design phase. It is our contention that system goals will be satisfied if every goal is associated with a role and every role is played by an agent class.

The general case transformation of goals to roles is one-to-one, with each goal mapping to a role. However, there are situations where it is useful to have a single role be responsible for multiple goals, including convenience or efficiency. One mapping of the goals from our previous example to a set of roles is shown below.

PaperDB	(1.1.1, 1.1.2, 1.1.2.1)
Partitioner	(1.2.1)
Assigner	(1.2.2)
Reviewer	(1.3.1)
Collector	(1.4)
DecisionMaker	(1.5, 1.5.1)

Due to the simplicity of our example, we mapped goals to individual roles with a two exceptions. Goals, 1, 1.1, 1.2, and 1.3 were not mapped to roles since they were partitioned. However, the PaperDB role was assigned all the goals associated with goal 1.1, namely 1.1.1, 1.1.2, and 1.1.2.1. In addition, the DecisionMaker role was assigned both 1.5 and 1.5.1, which are closely related.

Related goals can often be combined into a single role. For example, the "collect papers," "distribute papers," and "distribute abstracts" goals are combined into the single PaperDB role since they are closely related and require the same type of access techniques. While combining goals makes the role more complex, it may simplify the overall design.

In general, interfacing with external or internal resources requires a separate role to act as an interface to the rest of the system. We generally consider

a human user as an external resource. In MaSE we do not explicitly model human-computer interaction; we create a specific role to encapsulate the user interface. In this way, we can define the ways in which a user can interface with the system without defining the user interface itself. Other resources such as databases, files or legacy systems may also require their own interface role. In our example, the Author role does not satisfy any system goals as it is an interface to the user; however, without it, the system is not needed.

Role definitions are captured in a MaSE Role Model as shown in Figure 6.2, which includes information on interactions between role tasks and is more complex than traditional role models, as described in (Kendall, 1998). Roles are denoted by rectangles, while a role's tasks are denoted by ovals attached to the role. Lines between tasks denote communications protocols with the arrow pointing from the initiator to the respondent. Solid lines indicate external communications while dashed lines denote communication between tasks in the same role instance.

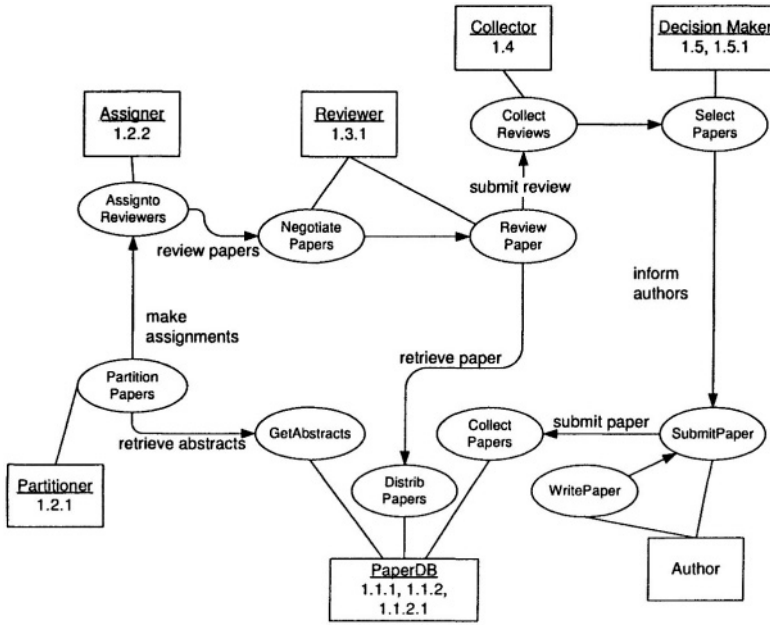

Figure 6.2. MaSE Role Model

The tasks are generally derived from the goals for which a task is responsible. For instance, the PaperDB role is responsible for attaining goals 1.1.1, 1.1.2, and 1.1.2.1. Therefore, to accomplish this goal, the role must be able to collect papers and distribute them and their abstracts. Therefore, we created three interrelated tasks: Collect Papers, Distrib Papers, and GetAbstracts.

While we could have specified all three goals in a single task, partitioning them in this was is modular and effectively encapsulates the actual approach used.

Roles should not share or duplicate tasks. Sharing of tasks is a sign of improper role decomposition. Shared tasks should be placed in a separate role, which can be combined into various agent classes in the Design phase.

Concurrent Task Model. After roles are created and tasks identified, the developer captures the role's behavior by defining the details of the individual tasks. A role may consist of multiple tasks that, when taken together, define the required behavior of that role. Each task executes in its own thread of control, but may communicate with each other. Concurrent tasks are defined in Concurrent Task Models (see Figure 6.3) and are specified as finite state automata, which consist of states and transitions. *States* encompass the processing that goes on internal to the agent while *transitions* allow communication between agents or between tasks.

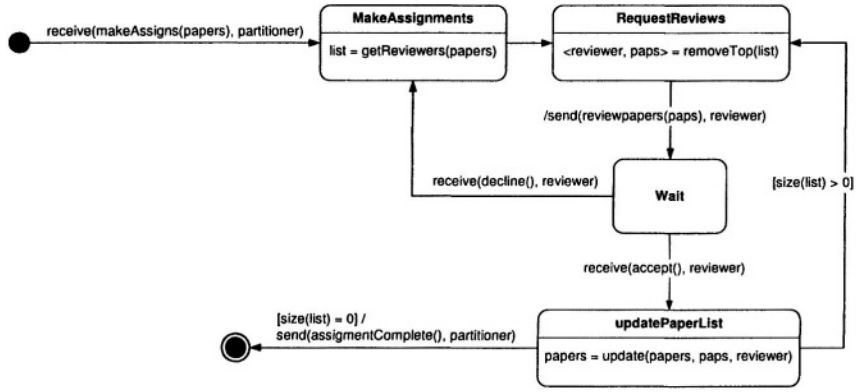

Figure 6.3. Concurrent Task Diagram

A transition consists of a source state, destination state, trigger, guard condition, and transmissions and uses the syntax *trigger [guard]* ^ *transmission(s).* Multiple transmissions may be separated with a semicolon (;), however, no ordering is implied. Generally, events on triggers or transmissions are to be associated with a task within the same role, thus allowing internal task coordination. However, two special events, send and receive, are used to indicate messages sent between agents. The *send* event (denoted *send(message, agent)*) is used to send a message to another agent while the *receive* event (denoted as *receive(message, agent)*) signifies the receipt of a message. The *message* is defined as a performative, which describes the intent of the message, along with a set of parameters that are the content of the message (i.e., *performa-*

tive(p1 ... pn) where *p1 ...pn* denotes *n* parameters). It is also possible to send a message to a group of agents via multicasting using a <group-name> versus a single agent name.

States may contain *activities* that represent internal reasoning, reading a percept from sensors, or performing actions via actuators. Multiple activities may be included in a single state and are performed in an uninterruptable sequence. Once in a state, the task remains there until the activity sequence is complete. The variables used in activity and events definitions are visible within the task, but not outside of the task or within activities. All messages sent between roles and events sent between tasks are queued to ensure that all messages are received even if the agent or task is not in the appropriate state to handle the message or event immediately.

Concurrent tasks have predefined activities to deal with mobility and time. The *move* activity specifies that the agent is to move to a new address and returns a Boolean value (*Boolean = move(location)*), which states whether the move actually occurred. The agent can reason over this value and deal with it accordingly.

To reason about time, the Concurrent Task Model provides a built in timer activity. An agent can define a timer using *t = setTimer*(*time*), the setTimer activity. The *setTimer* activity takes a time as input and returns a tinier that will timeout in exactly the time specified. The timer that can then be tested via the timeout activity, which returns a Boolean value, to see if it has "timed out" *(Boolean = timeout*(*t*)*).*

Once a transition is enabled, it is executed instantaneously. If multiple transitions are enabled, the following priority scheme is used.

1 Transitions whose triggers are internal events.

2 Transitions whose transmissions are internal events.

3 Transitions whose trigger receives a message from another role.

4 Transitions whose transmissions are a message to another role.

5 Transitions with valid guard conditions only.

Figure 6.3 shows the *Assign to Reviewers* task for the Assigner role. The task is initiated upon receipt of a makeAssigns message from a Partitioner agent, which includes a list of papers to be assigned. After the message is received, the task goes to the MakeAssignments state where it computes a list of reviewers for the papers (a process that is as yet undefined). Once these list is defined, the task transitions to the RequestReviews state where the top reviewer/papers tuple is taken off the list. A reviewPapers message is then sent to the reviewer effectively requesting that the agent provide a review for the

associated papers, which is denoted by the *paps* parameter. The task remains in the Wait state until a reply from the reviewer is received. If the reviewer declines (via a decline message), the task returns to the MakeAssignment state where it computes a new list of reviewers for the remaining papers. If the reviewer accepts the request via an accept message, the task transitions to the updatePaperList state where the list of papers is updated by adding the name of the reviewer to the papers that they will be reviewing. If the list is not empty, the task returns to the RequestReviews state to make a request of the next reviewer on the list. If the size of the reviewers list is empty, the task ends by sending an assignmentComplete message to the Partitioner agent.

3.4 Analysis Phase Summary

Once the concurrent tasks of each role are defined, the Analysis phase is complete. The MaSE Analysis phase is summarized as follows:

1 Identify goals and structure them into a Goal Hierarchy Diagram.

2 Identify Use Cases and create Sequence Diagrams to help identify roles and communications paths.

3 Transform goals into a set of roles.

(a) Create a Role Model to capture roles and their tasks.

(b) Define role behavior using Concurrent Task Models for each task.

4. Design Phase

There are four steps to the designing a system with MaSE. The first step is Creating Agent Classes, in which the designer assigns roles to specific agent types. In the second step, Constructing Conversations, the conversations between agent classes are defined while in the third step, Assembling Agents Classes, the internal architecture and reasoning processes of the agent classes are designed. Finally, in the last step, System Design, the designer defines the number and location of agents in the deployed system.

4.1 Creating Agent Classes

In the Creating Agent Classes step, agent classes are created from the roles defined in the Analysis phase. This phase produces an Agent Class Diagram, which depicts the overall agent system organization consisting of agent classes and the conversations between them. An *agent class* is a template for a type of agent in the system and are defined in terms of the roles they will play and the conversations in which they may participate. If roles are the foundation of MAS design, then agent classes are the bricks used to implement MAS.

These two different abstractions manipulate two distinct system dimensions. Roles allow us to allocate system goals while agent classes allow us to consider communications and other resource usage.

The first step is to assign roles to each agent class. If assigned multiple roles, agent classes may play them concurrently or sequentially. To ensure that system goals are accounted for, each role must be assigned to at least one agent class. The analyst can easily change the organization and allocation of roles among agent classes during design, since roles can be manipulated modularly. This allows consideration of various design issues, which are based on standard software engineering concepts such as functional, communicational, procedural, or temporal cohesion.

During this step, we also identify the conversations in which different agent classes must participate. An agent's conversations are derived from the external communications of the agent's assigned roles. For instance, if roles A and B communicate with each other, then, if agent 1 plays role A and agent 2 plays role B, then there must be a conversation between agent 1 and agent 2.

The agent classes and conversations are documented via Agent Class Diagrams, which are similar to object-oriented class diagrams with two main differences. First, agent classes are defined by the roles they play, not by attributes and methods. Second, all relationships between agent classes are captured as conversations. A sample Agent Class Diagram is shown in Figure 6.4. The boxes in Figure 6.4 denote agent classes and contain the class name and the set of roles each agent plays. Lines with arrows identify conversations and point from the conversation initiator to the responder. In this design, the PC Chair agent plays the Partitioner, Collector, and Decision Maker roles while the PC Member agent plays both the Assigner and Reviewer roles. Outside of Authors, the only other agent is the DB agent, which provides an interface to the database containing papers, abstracts, and author information.

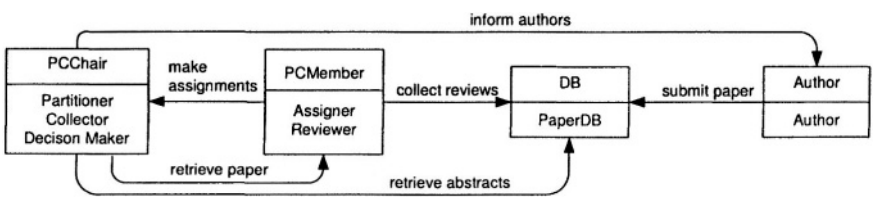

Figure 6.4. Agent Class Diagram

The Agent Class Diagram is the first design object in MaSE that depicts the entire MAS in its final form. If we have carefully followed MaSE to this point, the system represented by the Agent Class Diagram will support the goals and Use Cases identified in the Analysis phase. Of particular importance at this

point is the system organization – the way that the agent classes are connected with conversations.

4.2 Constructing Conversations

Constructing Conversations is the next MaSE Design phase step. So far, the designer has only identified conversations; the goal of this step is to define the details of those conversations based on the internal details of concurrent tasks.

A *conversation* defines a coordination protocol between two agents and is documented using two Communication Class Diagrams, one each for the initiator and responder. A Communication Class Diagram, as shown in Figure 6.5, is similar to a Concurrent Task Model and defines the conversation states of the two participant agent classes. The initiator begins the conversation by sending the first message. When the other agent receives the message, it compares it to its active conversations. If it finds a match, the agent transitions the appropriate conversation to a new state and performs any required actions or activities from either the transition or the new state. Otherwise, the agent assumes the message is a new conversation request and compares it to the conversations it can participate in with the sending agent. If the agent finds a match, it begins a new conversation.

As stated above, communication class diagrams use states and transitions to define the inter-agent communication. Transitions use the following syntax: *rec-mess(args1) [cond] / action ^ trans-mess(args2).* This states that if the message *rec-mess* is received with the arguments *args1* and the condition *cond* holds, then the method *action* is called and the message *trans-mess* is sent with arguments *args2.*

The transition from the start state in Figure 6.5 (left) indicates that it is the initiator half of a conversation, since it transmits a message. The conversation describes how the PCChair agent (the conversation initiator) sends a message to the Author agent notifying it of the acceptance. At this point, the PCChair enters a wait state. If the Author can still attend the conference, it sends an accept message (Figure 6.5 right) and the conversation is completed. If the Author cannot attend the conference, it returns a decline message. After receiving a decline message, the PCChair performs the updatePapers activity to update its list of attendees.

As discussed above, the designer establishes an agent's set of conversations by the roles it has been assigned. In the same way, the conversation design is derived from the concurrent tasks associated with those roles. Since a concurrent task integrates inter- and intra-role interactions, it provides the information required to define conversations. Each task that defines external communication creates one or more conversations. If all task communication is with a single role, or set of roles that have all been mapped to a single agent class, the

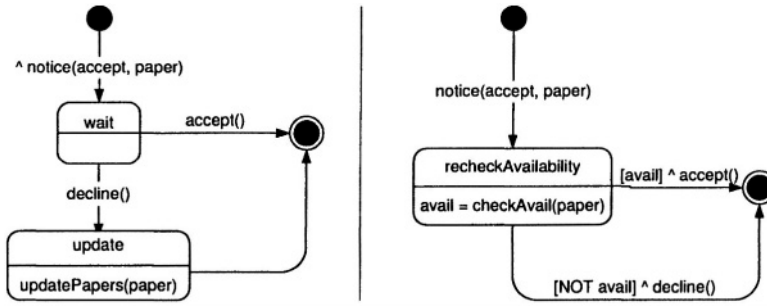

Figure 6.5. Inform authors conversation initiator and responder

task can be mapped directly to a single conversation. More generally, however, concurrent tasks spawn multiple conversations.

Once the information from Concurrent Task Models has been integrated into conversations, the designer must ensure that other factors, such as robustness and fault tolerance, are taken into account. For instance, if a particular agent sends a message to another agent requesting an action be performed, the conversation should be able to handle the other agent's refusal or inability to complete the request.

4.3 Assembling Agents

Agent class internals are designed during the step Assembling Agents, that includes two sub-steps: defining the architecture of agents and defining the architecture's components. Designers have the choice of either designing their own architecture or using predefined architectures such as BDI. Likewise, a designer may use predefined components or develop them from scratch. Components consist of a set of attributes, methods, and possibly a sub-architecture.

An example of an Agent Architecture Diagram is shown in Figure 6.6. Architectural components (denoted by boxes) are connected to either inner- or outer-agent connectors. *Inner-agent connectors* (thin arrows) define visibility between components while *outer-agent connectors* (thick dashed arrows) define external connections to resources such as agents, sensors and effectors, databases, and data stores. Internal component behavior may be represented by formal operation definitions or state-diagrams. The architecture and internal definition of the components must be consistent with the conversations defined in the previous step. At a minimum, this requires that each action or activity defined in a Communication Class Diagram be defined as an operation in one of the internal components. The internal component state diagrams and operations can also be used to initiate and coordinate various conversations.

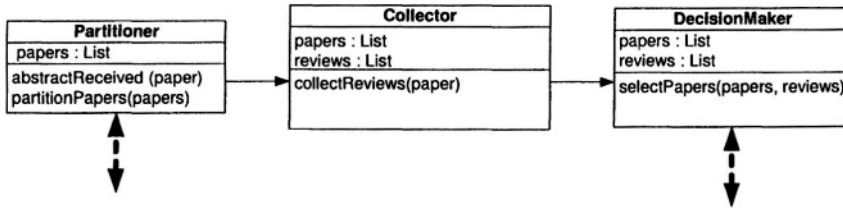

Figure 6.6. PCChair Agent Architecture

The PCChair agent architecture is shown in Figure 6.6. The PCChair agent has three components, which basically implement a pipeline architecture. The Partitioner component receives abstracts and uses the partitionPapers method to break the list into sets based on content. The Partitioner then calls the collectReviews method of the Collector component, which waits and collects all the reviews from the reviewer. Once all papers have been reviewed, the Collector component calls the selectPapers method of the DecisionMaker component, who selects the best papers and notifies the authors.

4.4 System Design

System Design is the final step of the MaSE methodology and uses Deployment Diagrams to show the numbers, types, and locations of agent instances in a system. System design is actually the simplest step of MaSE, as most of the work was done in previous steps. Figure 6.7 shows a Deployment Diagram for the conference management system. The three-dimensional boxes represent agents while the connecting lines represent actual conversations between agents. The agents are identified by their class name in the form of *instance-name : class.* Dashed boxes define physical computational platforms.

A designer should define the system deployment before implementation since agents typically require Deployment Diagram information, such as a hostname or address, for communications. Deployment Diagrams also offer an opportunity for the designer to tune the system to its environment to maximize available processing power and network bandwidth. In some cases, the designer may specify a particular number of agents in the system or the specific computers on which certain agents must reside. The designer should also consider the communication and processing requirements when assigning agents to computers. To reduce communications overhead, a designer may choose to deploy agents on the same machine. However, too many agents on a single machine destroys the advantages of distribution gained by using the multiagent paradigm. Another strength of MaSE is that a designer can make these mod-

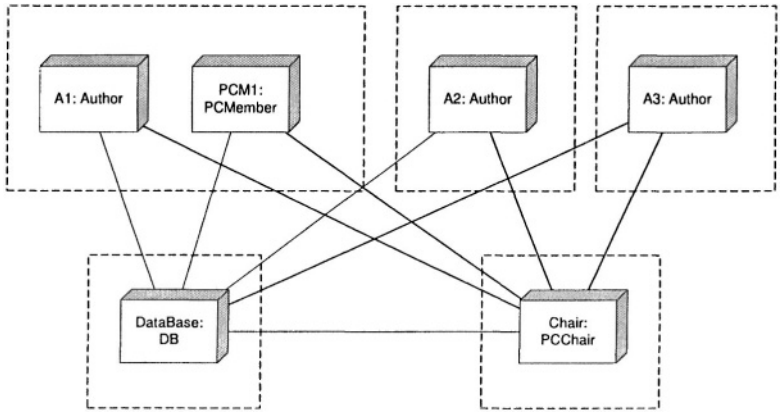

Figure 6.7. Deployment Diagram

ifications after designing the system organization, thus generating a variety of system configurations.

4.5 Design Phase Summary

Once the Deployment Diagrams are finished, the Design phase is complete. The MaSE Design Phase can be summarized as follows:

1 Assign roles to agent classes and identify conversations.

2 Construct conversations, adding messages/states for robustness.

3 Define internal agent architectures.

4 Define the final system structure using Deployment Diagrams.

5. agentTool

The agentTool system (DeLoach and Wood, 2001) has been developed to support and enforce MaSE. Currently agentTool implements all seven steps of MaSE as well as automated design support. The agentTool user interface is shown in Figure 6.8. The menus across the top allow access to several system functions, including analysis to design transformations (Sparkman et al., 2001), conversation verification (Lacey et al., 2000), and code generation. The buttons on the left add specific items to the diagrams while a text window 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.

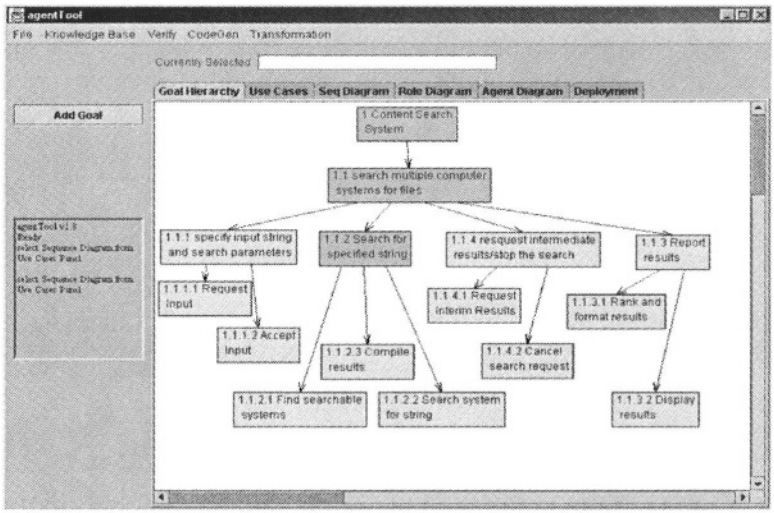

Figure 6.8. agentTool

While the designer may use existing architectures or design a new one from scratch, agentTool also provides the ability to semi-automatically derive the agent architecture directly from the roles and tasks defined in the analysis phase. This approach has the advantage of providing a direct mapping from analysis to design. Each task from each role played by an agent defines a component in the agent class. The concurrent task itself is transformed into a combination of the component's internal state diagram and a set of conversations. Activities identified in the concurrent task become methods of the component.

The transformation is actually a sequence of transformations that incrementally change roles and tasks into agent classes, components, and conversations. Before beginning the analysis-to-design transformation process, the Role Model and its set of concurrent tasks, and the assignment of roles to agent classes must exist. During the first stage of the transformation process, agentTool derives agent components from their assigned roles and assigns external events to specific protocols. In the second stage, agentTool annotates the component state diagrams to determine where conversations start and end. During the last stage, agentTool extracts the annotated states and transitions and uses them to create new conversations, replacing them in the component state diagram with actions initiating the conversation.

A second set of transformations that is currently implemented in agentTool consists of transformations to add functionality required for mobility. In the

analysis phase, mobility is specified using a *move* activity in the state of a concurrent task diagram. This *move* activity is copied directly into the associated component state diagram during the initial set of analysis-to-design transformation described above. During the mobility transformation, the existing design is modified to coordinate the mobility requirements between all components in the agent design. In the derived mobility design, the Agent-Component is responsible for coordinating the entire move and working with the external agent platform to save its current state and actually carry out the move.

The agentTool system also provides automatic verification of conversations. The verification process begins with the fully automated translation of system conversations into the Promela modeling language. Then, the Promela model is automatically analyzed using the Spin verification tool to detect errors such as deadlock, non-progress loops, syntax errors, unused messages, and unused states (Holzmann, 1997). Feedback is provided to the designer automatically via text messages and graphical highlighting of error conditions.

6. Applications

MaSE has been successfully applied in many graduate-level projects as well as several research projects. The Multiagent Distributed Goal Satisfaction project used MaSE to design the collaborative agent framework to integrate different constraint satisfaction and planning systems. The Agent-Based Mixed-Initiative Collaboration project also used MaSE to design a MAS focused on distributed human and machine planning. MaSE has been used successfully to design an agent-based heterogeneous database system as well as a multiagent approach to a biologically based computer virus immune system. More recently, we applied MaSE to a team of autonomous, heterogeneous search and rescue robots (DeLoach et al., 2003). The MaSE approach and models worked very well. The concurrent tasks mapped nicely to the typical behaviors in robot architectures. MaSE also provided the high-level, top-down approach missing in many cooperative robot applications.

7. Comparison with other Methodologies

There have been several methodologies proposed for developing MAS (see chapter 7). However, we only compare MaSE against the two other methodologies presented in this book: Gaia (see chapter 4) and Tropos (see chapter 5).

The Gaia method, as presented in chapter 4, is one of the best-known approaches to building MAS and has many similarities with MaSE. As in MaSE, Gaia uses roles as building blocks. In general, both the analysis phases of MaSE and Gaia capture much of the same type of information, although in different types of models. The major difference is in the level of support for

detailed agent design provided by Gaia. Gaia produces a high-level design and assumes the details will be developed using traditional techniques whereas MaSE provides models and guidance on creating the detailed design.

Tropos, which was presented in chapter 5, take a significantly different approach than MaSE. Tropos focuses on early requirements definition, which is not stressed with MaSE. Tropos uses Yu's *i** framework (Yu, 2001), which provides a nice front end to Tropos. In fact, the Tropos early requirements approach could be used with MaSE as the goal model of each methodology are essentially the same.