Intelligence Integration in Distributed Knowledge Management

Dariusz Król  
*Wrocław University of Technology, Poland*

Ngoc Thanh Nguyen  
*Wrocław University of Technology, Poland*
Chapter XIII
Multi-Agent Systems Engineering: An Overview and Case Study

Scott A. DeLoach
Kansas State University, USA

Madhukar Kumar
Software Engineer, USA

ABSTRACT

This chapter provides an overview of the Multi-agent Systems Engineering (MaSE) methodology for analyzing and designing multi-agent systems. MaSE consists of two main phases that result in the creation of a set of complementary models that get successively closer to implementation. MaSE has been used to design systems ranging from a heterogeneous database integration system to a biologically based, computer virus-immune system to cooperative robotics systems. The authors also provide a case study of an actual system developed using MaSE in an effort to help demonstrate the practical aspects of developing systems using MaSE.

INTRODUCTION

This chapter describes the Multi-agent Systems Engineering (MaSE) methodology for analyzing and designing multi-agent systems. MaSE was originally designed to develop closed, general purpose, heterogeneous multi-agent systems. MaSE has been used to design systems ranging from a heterogeneous database integration system to a biologically based, computer virus-immune system to cooperative robotics systems. While the multi-agent systems designed by MaSE are typically closed (the number and type of all agents are known a priori), the number of agents is unlimited, although, practically, the number of types of different agents is limited to something less than 50.
MaSE uses the abstraction provided by multi-agent systems to help designers develop intelligent, distributed software systems. MaSE views agents as a further abstraction of the object-oriented paradigm where agents are a specialization of 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. Agents are a convenient abstraction that allows designers to handle intelligent and non-intelligent system components equally within the same framework.

MaSE builds on existing object-oriented techniques and applies them to the specification and design of multi-agent systems. Many of the models developed with MaSE are similar to models defined in the Unified Modeling Language. However, the semantics of the models are often specialized for the multi-agent setting.

MaSE was designed to be used to analyze, design, and implement multi-agent systems by proceeding in an orderly fashion through the development lifecycle (DeLoach, Wood, & Sparkman, 2001). MaSE has been automated via an analysis and design environment called agentTool, which is a tool that supports MaSE and helps guide the system designer through a series of models, from high-level goal definition to automatic verification, semi-automated design generation, and finally to code generation.

The MaSE methodology consists of two main phases that result in the creation of a set of complementary models. The phases and the respective models that result at the end of each phase are listed below. While presented sequentially, the methodology is, in practice, iterative. The intent is to free the designer 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.

**ANALYSIS PHASE**

The first phase in developing a multi-agent system using the MaSE methodology is the analysis phase. The goal of the MaSE analysis phase is to define a set of roles that can be used to achieve the system-level goals. These roles are defined explicitly via a set of tasks, which are described by finite state models. This process is captured in three steps: capturing goals, applying use cases, and refining roles.

**Capturing Goals**

The purpose of the first step in the analysis phase is to capture goals of the system by extracting the goals from a set of system requirements. The initial system requirements may exist in many forms including informal text and tell the designer about how the system should function based on specific inputs and the system state. The MaSE methodology uses these requirements to define goals in two specific sub-steps: Identifying goals and Structuring goals.

<table>
<thead>
<tr>
<th>Phases</th>
<th>Models</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Analysis Phase</td>
<td>Goal Hierarchy</td>
</tr>
<tr>
<td>a. Capturing Goals</td>
<td>Use Cases, Sequence Diagrams</td>
</tr>
<tr>
<td>b. Applying Use Cases</td>
<td>Concurrent Tasks, Role Model</td>
</tr>
<tr>
<td>c. Refining Roles</td>
<td>Agent Class Diagrams</td>
</tr>
<tr>
<td>2. Design Phase</td>
<td>Conversation Diagrams</td>
</tr>
<tr>
<td>a. Creating Agent Classes</td>
<td>Agent Architecture Diagrams</td>
</tr>
<tr>
<td>b. Constructing Conversations</td>
<td>Deployment Diagrams</td>
</tr>
<tr>
<td>c. Assembling Agent Classes</td>
<td></td>
</tr>
<tr>
<td>d. System Design</td>
<td></td>
</tr>
</tbody>
</table>
Identifying Goals

The main purpose of this step is to derive the overall system goal and its subgoals from the initial set of requirements. This is done by first extracting scenarios from the requirements and then identifying the goals of the scenarios. These initial scenarios are usually abstract in nature and are critical to the entire system. Therefore, the goals identified from these scenarios are at a very high level. These high-level goals then serve as the basis of analysis of the entire system. The roles defined later in the analysis phase must support one of these goals. Later, if the analyst defines a role that does not support one of these goals, either the role is not needed or the initial set of goals was incomplete and a new goal must be added.

Structuring Goals

After the goals have been identified, the second step is to categorize and structure them into a goal tree. The result is a Goal Hierarchy Diagram whose nodes represent goals and arcs define goal/subgoal relationships. The Goal Hierarchy Diagram is acyclic; however, there some subgoals that may have more than one parent goal.

To structure the goals, the analyst first identifies the main goal of the system. In the case where there is more than one main goal, those goals must be summarized as one high-level goal that is decomposed into a set of subgoals that are easier to manage and understand. To decompose a goal into subgoals, the developer must analyze what must be done to achieve the parent goals. A subgoal should support its parent goal by describing a subgoal that must be achieved in order to achieve the parent goal.

Although superficially similar, goal decomposition is different from function decomposition since goals define what tasks must be done instead of how a task is achieved, which is functional decomposition. Thus, goal decomposition should stop when the designer thinks that any further decomposition will result in functions and not subgoals. MaSE goal decomposition is similar to the KAOS approach (van Lamsweerde & Letier, 2000) except that MaSE goals do not have to be strictly AND-refined or OR-refined.

There are four types of goals in a Goal Hierarchical Diagram: summary goals, partitioned goals, combined goals, and non-functional goals. Any goal or subgoal can take on the attributes of any one, or more, of these types of goals. The four types of goals are described below.

1. **Summary Goal.** A summary goal encapsulates a set of existing “peer” goals to provide a common parent goal for the set. This often happens at the highest level of the Goal Hierarchical Diagram when a goal may be needed to support multiple high-level goals.

2. **Non-Functional Goal.** As the name suggests, non-functional goals are derived from non-functional requirements of the system, such as maintaining reliability or response times of the system. These goals need not directly support the overall functional goals of the system. When a non-functional goal is discovered, a new branch is generally created under the overall system goal, which can then be decomposed into either functional or non-functional sub-goals.

3. **Combined Goal.** While analyzing the goals of a system, often a number of subgoals are discovered in a hierarchy that are identical or very similar and can be grouped into a combined goal. This often results when the same basic goal is a subgoal of two different goals. In this case, the combined goal becomes a subgoal of both the goals.

4. **Partitioned Goal.** A partitioned goal is one of a set of goals that collectively meet a parent goal. This is identical to the notion of a KAOS conjunctive goal.
Once the goals have been identified and structured, the developer is ready to move to the next step of the MaSE analysis phase, applying use cases.

**Applying Use Cases**

In this step, the goals and subgoals are translated into use cases. These use cases typically capture the scenarios discovered in the previous step by providing a textual description and a set of sequence diagrams that are similar to the UML sequence diagrams. The main difference between MaSE sequence diagrams and UML is that in MaSE they are used to represent sequences of events between roles instead of objects. The events sent between roles are used in later steps to help define the communications between the agents that will be eventually playing these roles.

The use case at this stage helps the developer in representing desired system behaviors and sequences of events. When the use cases are converted to sequence diagrams, the roles that are identified become the initial set of roles that will be used in the next step of refining roles.

While not all requirements can be captured as use cases, the developer should try to represent the critical requirements as either positive or negative use cases. Positive use cases define the desired system behaviors, and negative use cases describe a breakdown or an error in the system. Both are useful in defining roles that must be played in the system.

**Refining Roles**

With the Goal Hierarchy Diagram and use cases in place, the analyst is ready to move to the next step, Refining Roles. This step involves further defining roles by associating them with specific tasks. The roles produced from this step are defined in such a way as to ensure that each system goal is accounted for and form the building blocks for the agents that will eventually populate the system.

MaSE is built on the assumption that the system goals will be satisfied if each goal maps to a role, and every role is played by at least one agent class. In general, the mapping of goals to roles involves a one-to-one mapping. However, the developer may choose to allow a role to be responsible for multiple goals for the sake of convenience or efficiency. At this stage, the developer may also choose to combine several roles; although this will most certainly increase the complexity of the individual roles, it can significantly simplify the overall design.

In MaSE, role refinement is captured in a Role Model (Kendall, 1998). In this model, the roles are represented by a rectangle, while a role’s tasks are represented by ovals attached to them. The arrows between tasks designate communication protocols, with arrows pointing from the initiator of the protocol toward the responder. Solid lines represent external communication (role-to-role), while dashed lines indicate internal communication between tasks belonging to the same role instance.

Once the roles are decomposed into a set of tasks, the individual tasks are designed to achieve the goals for which the role is responsible. It is important to note here that roles should not share tasks with other roles. Sharing a task among different roles indicates improper role decomposition. If the analyst believes that a task needs to be shared, then a separate role should be created for that task. This will allow the task to be incorporated into different agent classes, thus being effectively shared.

**Concurrent Task Model**

After the roles are defined, the analyst must define the details of each task in the role model. Task definition is performed via a Concurrent Task Diagram, which is based on finite state automata. Semantically, each task is assumed to run concurrently and may communicate with other tasks either internally or externally. Taken collectively,
the set of tasks for a specific role should define the behavior required for that role.

A concurrent task consists of a set of states and transitions. The states in the concurrent tasks represent the internal functioning of an agent while transitions define the communication between tasks. Every transition in the model has a source state, destination state, trigger, guard condition, and transmissions. The transitions use the syntax

\[
\text{trigger [guard] } \rightarrow \text{transmission(s)}
\]

If there are multiple transmissions required, they can be concatenated using a semicolon (;) as a separator; however, no ordering is implied. In general, events sent as triggers or transmissions are associated with events sent to tasks within the same role instance, thus allowing for internal task coordination. To represent messages sent between agents, however, two special events—send and receive—are used.

The send event is used to represent a message sent to another agent and is denoted by send(message, agent) while a receive event, denoted by receive(message, agent), is used to define a message received from another agent. The message itself consists of a performative, the intent of the message along with a set of parameters. It is also possible to send the same message to several agents at the same time using multicasting by using a group name of the agents as compared to the name of a single agent.

Task states may contain activities that represent internal reasoning, reading a percept from sensors, or performing actions via actuators. More than one activity may be included in a single state and they are performed in an uninterruptible sequence, which, when combined with states and transitions, gives a general computational model. Once inside a state, the task remains there until the activity sequence is complete. Variables used in activity and event 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.

Once a transition is enabled, it is executed instantaneously. If multiple transitions are enabled, then internal events are handled first, external messages (the send/receive events) are next, and the transitions with guard conditions only are last (DeLoach, 2000).

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

**DESIGN PHASE**

In the analysis phase, a set of goals was derived and used to create a set of use cases and sequence diagrams that described basic system behavior. These models were then used to develop a set of roles and tasks that showed how the goals should be achieved. The purpose of the design phase is to take those roles and tasks and to convert them into a form that is more amenable to implementation, namely, agents and conversations. The MaSE design phase consists of four steps. These steps include designing agent classes, developing conversation between the agents, assembling agents, and finally deploying the agents at system-level design.

**Construction of Agent Classes**

The first step in the design phase involves designing the individual agent classes, which is documented in an Agent Class Diagram. In this
step, the designer maps each role defined in the analysis phase to at least one agent class. Since roles are derived from the system goals and are responsible for achieving them, enforcing the constraint that each role is assigned to at least one agent class in the system helps to ensure that the goals are actually implemented in the system. In general, an agent class can be thought of as a template for creating the actual agent instances that will be part of the multi-agent system. These templates are defined in terms of the roles they play and the protocols they use to coordinate with other agents.

The first step in constructing agent classes is to assign roles to the agent classes. If the designer chooses to assign more than one role to the same agent class, the roles may be performed either concurrently or sequentially. The assignment of roles to agents allows the multi-agent organization to be easily modified, since the roles can be manipulated modularly. This allows the designer to manipulate the design to account for various software engineering principles, such as functional or temporal cohesion.

Once the agents are created by identifying the roles they will be playing, the conversations between agents are designed accordingly. For example, if two roles, R1 and R2, that shared a communication protocol were assigned to agent classes A1 and A2 respectively, then A1 and A2 would require a conversation (to implement the protocol) between them as well.

The Agent Class Diagram that results from this step is similar to object-oriented class diagrams. They are different in that (1) agent classes are defined by the roles they play instead of their attributes and methods, and (2) the relationships between agent classes are always conversations.

Converting Conversations

Once the agent classes have been defined and the required conversations identified, the detailed design of the conversations is undertaken. These details are extracted from the communications protocols identified in the analysis phase.

Conversations are modeled using two different Conversation Class Diagrams, one for the initiator and the other for the responder. These diagrams are based on finite state automata and use states and transitions to define the inter-agent communication, similar to concurrent tasks. The transitions in the conversation diagrams use a slightly different syntax

\[ \text{rec-mess}(\text{args}1) \ [\text{cond}] / \text{action} \hat{\rightarrow} \text{trans-mess}(\text{args}2) \]

This means that if the message \( \text{rec-mess} \) is received with the arguments \( \text{args}1 \) and the condition \( \text{cond} \) holds true, then the method \( \text{action} \) is called and the message \( \text{trans-mess} \) is sent with arguments \( \text{args}2 \).

Conversations are derived from the concurrent tasks of the analysis phase, based on the roles the agents are required to play. Thus, each task that defines an external conversation (outside the role) ends up becoming one or more conversation between agents. However, if all task communication is internal (within the same role) or with roles that are performed by the same agent, then the communication translates into internal function or method calls. Generally, however, concurrent tasks translate into multiple conversations, as they require communication with more than one agent class.

During this stage, the designer also needs to take into account other factors besides the basic protocols defined in the concurrent tasks. For example, what should an agent do if it does not receive the message it was expecting? Perhaps the communication medium was disabled or the other agent failed. Therefore, the designer should attempt to make conversations robust enough to handle potential run-time errors.
Assembling Agent Classes and Deployment Design

The last two stages in MaSE involve the internal design of the agent classes and the system-level design. The first of these stages, Assembling Agent Classes, involves two steps, defining the agents’ architecture and defining the individual components of the architecture. MaSE does not assume any particular agent architecture and attempts to allow a wide variety of existing and new architectures to be used. Thus, the designer has the choice of either using pre-existing agent architecture like Beliefs, Desires, and Intentions (BDI) or creating a new architecture from scratch. The same goes for the architecture components. The step of assembling agents result in an Agent Architecture Diagram in which the components are represented by rectangular boxes connected to either inner or outer agent connectors. The inner-agent connectors, represented by thin arrows define visibility between components, while the outer agent connectors, represented by dashed arrows, define external connections to resources like other agents effectors, databases, and so on. A more detailed discussion of this step can be found in Robinson (2000).

The last step in building a multi-agent system using the MaSE methodology is to decide on the actual configuration of the system, which consists of deciding the number and types of agents in the system and the platforms on which they should be deployed. These decisions are documented in a Deployment Diagram, which is very similar to a UML Deployment Diagram and is used for much the same purpose. In a Deployment Diagram, agents are represented by three-dimensional boxes, while rectangles with dashed lines represent physical computing platforms. The lines between agents represent the actual lines of communication between the agents. In a dynamic multi-agent system in which agents move or are created and destroyed, the Deployment Diagrams are used to show snapshots of possible system configurations.

EXAMPLE CASE STUDY: MULTI-AGENT RESEARCH TOOL (MART)

To show how to use the MaSE methodology outlined above, this section presents an example of using MaSE to develop an actual multi-agent system. The Multi-agent Research Tool (MART) was developed as part of a MS project at Kansas State University and is being considered for distribution by a private company. The analysis and design was performed using the agentTool development environment, with the implementation being done in Java.

Overview

Writing articles is an important part of work for a researcher at a university or a content provider working for a media company. While writing research or news articles, the author often conducts searches on the World Wide Web (WWW) to unearth relevant information that can be used to write the article. However, when an author is writing an article, it is often a distraction to stop writing, visit a few search engines, conduct keyword searches, retrieve relevant information, and then incorporate it into the article. This is not very efficient, especially when the author has to deliver the article by a specific deadline.

The motivation for developing a Multi-agent Research Tool (MART) was to develop a tool that helps authors to research while writing an article without wasting valuable time. This means that the research tool should not only be smart and efficient in conducting searches, but that it should also be able to work in the background and, at the same time, be non-intrusive to the user. Moreover, since use of the Internet has become commonplace, it is assumed that a person using MART has access to the Internet. It would be more useful if the research tool could use distributed computing to retrieve research material and present it to the user whenever he/she decides to view or use them.
Based on the nature of the original motivation, it was decided to build MART as a multi-agent system since the location and numbers of the various components within the local network would not be known in advance. Since MART was developed using the MaSE methodology, the decision to use the agentTool development environment—a software engineering tool that directly supports MaSE analysis and design—was straightforward.

**Developing Goals of the System**

In the first step of the Analysis Phase, the following goals were defined based on the requirements for the MART system, as presented above. As shown in Figure 1, the overall goal of the system is to produce the results of a search for keywords from the user’s article. This goal is partitioned into four subgoals: ranking and refining the keywords used in the search, searching the Web for results, producing and presenting the result to the user, and managing the entire system.

The *rank and refine search keywords* goal is partitioned into two subgoals: reading user keywords and ranking the keywords. The goal, *search the Web*, is also partitioned; however, it has only one subgoal, namely, *search Web sites*. Although not technically required, this goal structure was adopted so that future versions could add additional goals that could include searching other types of information sources such as databases located on the host computer and/or local network. Finally, the goal *produce results* is partitioned into three subgoals that allow for reading raw results, refining the raw results, and producing the final results that will be presented to the user.

**Applying Use Case**

After defining the goals, three primary use cases were generated based on the three main subgoals (1.1, 1.2, and 1.3 in Figure 1). These use cases are *Refine and Rank Keywords*, *Search the Web*, and *Generate Results*. Each is presented in detail below.

**Refine and Rank Keywords**

The Refine and Rank Keywords use case defines how the system should behave when it is initially asked to perform a search. As shown in Figure 2, the manager of the search process asks the reader to read the predefined user preferences and keywords and then asks the ranker to rank the keywords that were returned. The user preferences define exactly how the user prefers the search to be conducted while the keywords are the specific words on which the user wants the search to be conducted. These keywords are then ranked in terms of relevance to the article the user is currently writing.

**Figure 1. MART system goals**

![Diagram of MART system goals]
Search the Web

As shown in Figure 3, the Search the Web use case defines the basic search process of the system once a set of keywords has been developed. Each searcher agent is asked to search its known Web sites for a specific set of keywords. Exactly where and how each searcher conducts its search is not known to the manager. However, once results are received back by the searcher agents, the results are returned to the controller, who tabulates the results for a variety of searchers.

Generate Results

The sequence diagram in Figure 4 shows that the manager, once it has the raw results, sends a message to the result generator along with the raw results. The result generator refines the results by extracting duplicates and providing proper formatting and then sends back the finished product back to the manager.

Refining Roles

The role diagram depicts how the different goals are mapped to the roles of the system. Figure 5 shows that the controller has many tasks that collaborate with the other roles in order to read keywords, perform a search, and generate the finished product. The numbers inside rectangles (roles) indicate the goals for which they are responsible. For example, the Controller role is responsible for goal 1.4 from Figure 1, which is the goal of managing the system.

As discussed above, the solid lines connecting the different roles represent the communication between the roles. The dotted line between tasks

Figure 2. Sequence diagram for “Refine and Rank Keywords” use case

Figure 3. Sequence diagram for the use case “Search the Web”
Multi-Agent Systems Engineering

in the Sleuth role (the makeRaw protocol) shows that it is an internal communication between tasks within the same role instance (agent). The makeRawResults task is invoked by the rawResults task of the Sleuth once it receives the searchTheWeb request from the controller.

Concurrent Task Model

Once the Role Model has been constructed, a concurrent task model was defined for each task in the role model. For example, showKeywords is a task for the KeywordsReader role. An example of a concurrent task model for showKeywords is shown in Figure 6. The task starts when a readPrefs message is received from an agent named controller. After receiving the message, the user preferences are read via the activity readPreferences() in the readPreferences state. Upon completion of the activity, the task enter the readKeywords state where it gets the keywords from the user via the readKeywords() activity. If the keywords list is empty (null), then the task ends without sending a response. Otherwise, the task sends the set of keywords back to the controller.

Constructing Agent Classes

After all the tasks from the Role Model have been defined via concurrent task diagrams, the analysis phase ended and the design phase commenced. The first step of the design phase is to define the basic system architecture using an Agent Class Diagram. The initial task was to create agent classes and assign them specific roles to play.
The Agent Class Diagram shown in Figure 7 shows that MART has five different agents: AgentManager, AgentKey, AgentProducer, AgentGoogle, and AgentTeoma. The lines connecting the agents represent the conversations between the agents. For example, searchTheWeb is a conversation that is initiated by AgentManager. searchTheWeb is unique in that it exists between the AgentManager and two different agent types: AgentGoogle and AgentTeoma. Actually, both of these agent types implement the Sleuth role, and the conversation can be directed from the AgentManager to either agent type requesting them to conduct a search and return raw results. The agent classes in Figure 7 represent independent processes operating in their own thread of control. These agents could be placed on different machines and still be able to talk to each other using the conversations defined in the system.

**Constructing Conversations**

After creating the agent classes and documenting them via the Agent Class Diagram, the individual communication between the agents was defined, based on the protocols between the appropriate roles from which they were derived. Each resulting conversation was documented using a pair of Conversation Diagrams, which are similar to and can be derived from the concurrent tasks models.
developed during the analysis phase (Sparkman, DeLoach, & Self, 2001). Each conversation is represented from the initiator’s and the responder’s points of view. For example, the conversation rankKeywords from the above diagram has the agent class AgentManager as the initiator and agent class AgentKey as the responder. The diagrams are show below in Figure 8 and Figure 9. Taken together, the diagrams show that the initiator sends the rankKey message with the parameter keywords to the responder and then waits until a rankedKey message is returned along with the ranked set of keywords via the message parameter. The responder side of the conversation is quite similar, with the messages sent by the initiator being received and the messages received by the initiator being sent. The obvious difference between the two is that the responder side includes an activity, rankKeywords(), that is called to actually perform the ranking process.

Assembling Agents and Deployment

After developing the conversations required for MART, the next step was constructing the individual components of the agent classes. As discussed earlier, there is a choice of either using either pre-existing agent architectures or creating an application-specific architecture. Because the MART agents were simple, it was decided that a simple application-specific architecture was the best approach for MART. Each concurrent task was mapped directly to an internal component in the architecture, thus making the internal agent design directly reflect the roles and tasks of the analysis model. An example of component structure of the AgentKey agent class is shown in Figure 10.

The attributes and methods of the showKeywords component are derived directly from the showKeywords task defined in Figure 6, with the exception of the conv_r_readUserPreferences method. The conv_r_readUserPreferences method was created to initiate the readUserPreferences conversation. When the agent wants to start the readUserPreferences conversation, it calls the method, which contains all the implementation dependent code for handling the conversation. The rankedKeywords component was derived similarly. Because these two components do not communicate directly (they are derived from the showKeywords and rankedKeywords tasks in Figure 5), there is not a visibility connection between them.

The last step in the design of the MART system was to develop the overall system deployment

Figure 8. Conversation model for rankKeywords Initiator

Figure 9. Conversation model for rankKeywords Responder
design. The MART Deployment Diagram, as shown in Figure 11, was created as an example of how the MART system could be deployed. As the MART system is designed to be deployed in a number of settings, this deployment is notional. Typically, the AgentKey and AgentManager agents are started on the user’s computer while the AgentGoogle and AgentTeoma agents must be pre-deployed on local network computers (the AgentProducer agent may be deployed anywhere on the local network as well).

**Refining the Object Model**

After completing the analysis and design for MART, the implementation of the system began. As the system analysis and design was performed using agentTool, the first step was to verify that the conversations were correct and deadlock free. After this step was completed, the code generation capabilities of agentTool were employed to generate the initial code, which included stubs for each agent, each component, and each side of the conversations. Generally, each agent component is implemented as a single class. However, due to the simplicity of the components defined during the Assembling agents step, each agent class design was implemented by integrating all the agent components into a single class (one class for each agent type). This resulted in the simplified class structure shown in Figure 12.

The system architecture is shown in Figure 13, where each agent class is represented as a package. Obviously, multiple versions of each agent may

**Figure 10. Components of AgentKey**

<table>
<thead>
<tr>
<th>showKeywords</th>
<th>rankedKeywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>+pref : type</td>
<td>+keywords : Set type</td>
</tr>
<tr>
<td>+ AgentKey : type</td>
<td>+controller : type</td>
</tr>
<tr>
<td>+keywords : type</td>
<td>+rank : type</td>
</tr>
<tr>
<td>+controller : type</td>
<td></td>
</tr>
<tr>
<td>#readPreferences() : Object</td>
<td>#readKeywords() : Object</td>
</tr>
<tr>
<td>#read(keywords: String, preferences: String) : Object</td>
<td>#conv_r(rankKeywords: String, controller: Object) : void</td>
</tr>
</tbody>
</table>

**Figure 11. Deployment diagram for MART**
Multi-Agent Systems Engineering

exist. The **KeyObjectPackage**, which is accessed by each of the agents, includes shared definitions of data objects passed between agents. The diagram also includes a user-defined stereotype «conversation» to denote the existence of conversations between the various agent packages. The system was implemented using the agentMom agent-oriented middleware system, which inherently supports the concept of conversations as defined in MaSE (Mekprasertvit, 2004).

**STRENGTHS AND WEAKNESSES**

**Strengths**

MaSE is a comprehensive methodology for building multi-agent systems. It has been used to develop both software multi-agent systems and multi-agent cooperative robotic systems (DeLoach, Matson, & Li, 2003). One of the major strengths of MaSE is that it provides guidance throughout the entire software development lifecycle—from requirements elicitation through to implementation.

Firstly, MaSE is independent of any particular agent architecture or underlying environment. While the example above does not use pre-existing agent architectures, MaSE does allow the task and conversation behavior to be implemented in any architecture the designer wishes. For example, Robinson (2000) has defined a variety of agent architectures using MaSE components including reactive, BDI (Belief Desires and Intentions), knowledge-based, and planner based.

The sequence of interrelated MaSE models allows the developer to track the mapping of entities from one model to the next. This is most

*Figure 12. Simplified UML class model for MART*
readily apparent in the mapping of goals to roles, roles to agents, and communications protocols to conversations. This mapping allows the developer to move between models, showing that the entities defined in previous models are implemented successfully in the current model. It also provides an excellent tool for tracking down system errors. If a particular goal is not being successfully achieved, the developer can track that goal directly to the responsible role and then on the implementing agent classes.

Often in multi-agent approaches, developers are allowed to specify behavior and agent communications protocols; however, the relationship between the two is not always clear. MaSE provides a way of directly defining the relationship between agent communication protocols and the internal behavior of the agent. This relationship is captured in the concurrent task diagrams and is carried over to agent conversation diagrams. By studying a set of concurrent tasks, it becomes evident how the communications between roles, and eventually agents, directly affects and is affected by the results of the communication. For instance, in Figure 6, it is clear that the computation (the readPreferences activity) starts after receiving the readPrefs message, and the results of the readKeywords activity determine whether the userKeywords message is even sent.

MaSE is also supported by the agentTool development environment. AgentTool is a software engineering tool built to help designers create multi-agent systems using the MaSE methodology. Using agentTool, a multi-agent system can be developed by following the MaSE steps in both the analysis and design phases. Since agentTool is a graphical-based tool, all the diagrams and the models described in the MaSE methodology are created using the tool. During each step of system development, the various analysis and design diagrams are available via agentTool and the developer is allowed to move freely back and forth between models in the various MaSE steps. A developer may also use agentTool to verify a conversation at any point by using the conversation verification capability (Lacey & DeLoach, 2000), which uses the Spin model checker (Holzmann, 1997) to check for deadlocks, as well as non-progress loops, syntax errors, unused messages, or unused states. If an error exists, the verification results are presented textually to the developer as well as by directly highlighting the offending conversation diagram. AgentTool includes developing support for semi-automatic transformations.
that convert a set of analysis models into the appropriate design models (Sparkman, DeLoach, & Self, 2001). To initiate the process, the designer assigns roles to specific agent classes and then applies the semi-automated transformations. There are three transformation stages. In stage one, the transformations determine to which protocol individual concurrent task events belong. Next, the transformations create internal components for each concurrent task associated with the agent class. In the final stage, the conversations are extracted from the concurrent tasks and placed in conversation diagrams.

**Weaknesses**

While MaSE provides many advantages for building multi-agent systems, it is not perfect. It is based on a strong top-down software engineering mindset, which makes it difficult to use in some application areas. First, MaSE is not currently appropriate for the development of open multi-agent systems. Since MaSE predefines the communications protocols between agent classes, the resulting system assumes that any agents trying to participate in the system implicitly know what those protocols are. In addition, MaSE does not inherently support the use of different ontologies, although an extension to MaSE by DiLeo, Jacobs, and DeLoach (2002) does incorporate the notion of ontologies into MaSE and agentTool. In general, however, MaSE implicitly defines an ontology that is embedded in the task communication protocols and is implemented within each agent.

The MaSE notion of conversations can also be somewhat bothersome, as they tend to decompose the protocols defined in the analysis phase into small, often extremely simple pieces when the original protocol involves more than two agents. This often results in conversations with only a single message. This makes comprehending how the individual conversations fit together more difficult.

MaSE also tends to produce multi-agent systems with a fixed organization. Agents developed in MaSE tend to play a limited number of roles and have a limited ability to change those roles, regardless of their individual capabilities. Recent trends in multi-agent systems are towards the explicit design and use of organizations, which allow heterogeneous agents to work together within well-defined roles to achieve individual and system-level goals. In multi-agent teams, the use of roles and goals allows the agents to perform their duties in an efficient and effective manner that allows the team to optimize its overall performance. In most multi-agent design methodologies, including MaSE, the system designer analyzes the possible organizational structure and then designs one organization that will suffice for most anticipated scenarios. Unfortunately, in dynamic applications—where the environment as well as the agents may change—a designer can rarely account for or even consider all possible situations.

Ideally, a multi-agent team would be able to design its own organization at runtime. To accomplish this, MaSE would have to be extended to be able to analyze and design multi-agent organizations. While MaSE already incorporates many of the required organizational concepts such as goals, roles, and the relations between these entities, it cannot currently be used to define a true multi-agent organization.

**CONCLUSION**

MaSE provides a detailed approach to the analysis and design of multi-agent systems. MaSE combines several established models into a comprehensive methodology. It also provides a set of transformation steps that shows how to derive new models from the existing models thus guiding the developer through the analysis and design process.
MaSE has been successfully applied in many graduate-level projects as well as several research projects. The Multi-agent 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 multi-agent system focused on distributed human and machine planning. MaSE has been used successfully to design an agent-based heterogeneous database system as well as a multi-agent 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, Matson, & Li, 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.

Future work on MaSE will focus on specializing it for use in adaptive multi-agent 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 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.

REFERENCES


This work was previously published in Agent-Oriented Methodologies, edited by B. Henderson-Sellers and P. Giorigin, pp. 317-340, copyright 2005 by IGI Publishing, formerly known as Idea Group Publishing (an imprint of IGI Global).