Applying Agent Oriented Software Engineering to Cooperative Robotics

Scott A. DeLoach

Department of Computing and Information Sciences, Kansas State University
234 Nichols Hall, Manhattan, KS 66506
(785) 532-6350
sdeloach@cis.ksu.edu

Abstract
This paper reports our progress in applying multiagent systems analysis and design techniques to autonomous robotics applications. In this paper, we apply the Multiagent Systems Engineering (MaSE) methodology to design a team of autonomous, heterogeneous search and rescue robots. MaSE provides a top-down approach to building multirobotic systems instead of the bottom up approach employed in most robotic implementations. We follow the MaSE steps and discuss various approaches and their impact on the final system design.

Introduction
There have been many advances in the area of agent-oriented software engineering recently; however, there are few examples of applying such approaches to cooperative robotic applications. While there have been a number of architectures developed for cooperative robotics, there have been few attempts at defining high-level approaches to systems design (Parker 1998). In this paper, we attempt to determine the applicability of multiagent design approaches to cooperative robotics. We believe that using multiagent approaches for cooperative robotics may provide some of the missing elements evidenced in many cooperative robotic applications, such as generality, adaptive organization, and fault tolerance (Parker 1996).

In this paper, we apply the Multiagent Systems Engineering (MaSE) methodology to design a team of autonomous, heterogeneous search and rescue robots. MaSE provides a top-down approach to building multirobotic systems instead of the bottom up approach employed in most robotic implementations. We follow the steps of the MaSE methodology and discuss various approaches and their impact on the final system design. We do assume that the low-level behaviors common to mobile robots, such as motion and sensor control, already exist in libraries. Our focus is on designing high-level cooperative behaviors for specific applications.

Designing Multirobotic Agent Systems
We chose to use the MaSE methodology (DeLoach, Wood and Sparkman 2001) to design our multirobot system because it provides a top-down approach and a detailed sequence of models for developing multiagent systems. The seven-step MaSE process is shown in Figure 1, where the rounded rectangles denote the models used in each step. The goal of MaSE is to guide a system developer from an initial system specification to system implementation. This is accomplished by directing the designer through this set of inter-related system models.

Figure 1. MaSE Methodology
Because MaSE was designed to be independent of any particular multiagent system architecture, agent architecture, programming language, or communication framework, it seemed a good fit for cooperative robotic design. The next few paragraphs briefly describe the steps of MaSE as applied to our system. However, because we are focusing on high-level design issues, we do not delve into the details of designing the internal agent architecture, which is captured in the Assembling Agent Classes step. However, MaSE does provide general capabilities for modeling various generic agent (robot) architectures, such as ALLIANCE (Parker 1996).
Capturing Goals

The first step in MaSE is Capturing Goals, which takes the initial system specification and transforms it into a structured set of system goals as depicted in a Goal Hierarchy Diagram (Figure 2). In MaSE, a goal is a system-level objective; agents may be assigned goals to achieve, but goals have a system-level context.

There are two steps to Capturing Goals: identifying the goals and structuring goals. The analyst identifies goals by analyzing whatever requirements are available (e.g., detailed technical documents, user stories, or formal specifications). Once the goals have been captured and explicitly stated, they are analyzed and structured into a Goal Hierarchy Diagram. In a Goal Hierarchy Diagram, goals are organized by importance. Each level of the hierarchy contains goals that are roughly equal in scope and sub-goals are necessary to satisfy parent goals. Eventually, each goal will be associated with roles and agent classes that are responsible for satisfying that goal.

Applying Use Cases

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

To help determine the actual communications required within a system, the use cases are restructured as Sequence Diagrams, as shown in Figure 3. A Sequence Diagram depicts a sequence of events between multiple roles and, as a result, defines the minimum communication that must take place between roles. The roles identified in this step form the initial set of roles used to fully define the system roles in the next step and the events identified are used later to help define tasks and conversations.

In the example in Figure 3, we assume a team consisting of four roles: one searcher, one organizer, and (at least) two rescuers. Basically, the sequence diagram shows a sequence of events that could happen when an agent playing the searcher role locates a victim. Basically the searcher informs an organizer, who in turn notifies all available rescuers. Each rescuer returns its cost to retrieve the victim (based on location, number of other victims to retrieve, etc.) to the organizer who selects the most appropriate rescuer. The organizer notifies the rescuers of their assignments for this victim and then notifies the original searcher that help is on the way. Note that just because we have identified an organizer role in the use case, we do not have to have an organizer agent in the final design. The organizer role can be assigned to any agent (robot) in the final design or even the environment if we are using an appropriate framework with the ability to perform reactive tasks (Murphy, Picco and Roman 2001).

Refining Roles

The third step in MaSE is to ensure we have identified all the necessary roles and to develop the tasks that define role behavior and communication patterns. Roles are identified from the use cases as well as the system goals. We ensure all system goals are accounted for by associating each goal with a specific role that is eventually played by at least one agent in the final design. Each goal is usually mapped to a single role. However, there are many situations where it is useful to combine multiple goals in a single role for convenience or efficiency. Roles definitions are captured in a standard Role Model as shown in Figure 4.

Figure 2. Goal Hierarchy Diagram

Figure 2 shows an initial high-level goal hierarchy for the robotic search and rescue domain. Obviously, this could be broken down into more specific goals that each agent could use in attaining these goals; however, the purpose of the goal hierarchy diagram in MaSE is to identify the main system level goals, not individual agent goals.

Figure 3. Sequence Diagram

Figure 3 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 and the events identified are used later to help define tasks and conversations.

In the example in Figure 3, we assume a team consisting of four roles: one searcher, one organizer, and (at least) two rescuers. Basically, the sequence diagram shows a sequence of events that could happen when an agent playing the searcher role locates a victim. Basically the searcher informs an organizer, who in turn notifies all available rescuers. Each rescuer returns its cost to retrieve the victim (based on location, number of other victims to retrieve, etc.) to the organizer who selects the most appropriate rescuer. The organizer notifies the rescuers of their assignments for this victim and then notifies the original searcher that help is on the way. Note that just because we have identified an organizer role in the use case, we do not have to have an organizer agent in the final design. The organizer role can be assigned to any agent (robot) in the final design or even the environment if we are using an appropriate framework with the ability to perform reactive tasks (Murphy, Picco and Roman 2001).

Figure 4. Role Model

In our search and rescue system, we have identified three roles: searcher, rescuer, and organizer. The role model can be used to explain high-level system operation. In Figure 4 we can see that searcher roles negotiate with each other to determine the areas each will explore. After the negotiation is complete, the searchers go to their assigned...
areas and attempt to locate victims. Once victims are located, they send the information to the organizer, who in turn attempts to find the appropriate rescuer to find and rescue the victims. The rescuers then carry out the rescue.

**Defining Tasks.** Once roles have been identified, the detailed tasks, that define how a role accomplishes its goals, are defined and assigned to specific roles. A set of concurrent tasks provide a high-level description of what a role must do to satisfy its goals, including how it interacts with other roles. This steps is documented in an expanded role model as shown in Figure 5. The ellipses in the diagram denote tasks performed by the attached role while the arrows between tasks define protocols that specify how communication is performed between roles. In our search and rescue system, the Searcher role has two basic tasks: (1) to find an area to search, which must be negotiated with other Searcher roles, and (2) to locate victims in its define search area. The dotted line protocol between the two tasks denotes an internal communication between tasks in the same agent whereas the solid lines represent communication between different agents.

![Figure 5. Role Model with Tasks](image)

An example of a **Concurrent Task Diagram** defining the Locate Victim task is shown in Figure 6. The syntax of state transitions is trigger(args1) guard / transmission(args2), which means 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). Actions within each state are executed sequentially and are written as functions.

Locate Victim is a **reactive task**, which means that it is initiated whenever a search(area) message is received from the Find Area to Search task. After the task receives a search area, it plans a route to get to the area and then goes about executing the route. If route execution fails, the task re-plans the route and updates the map. When the robot gets to its area, it scans the area for victims. If one is found, it notifies an organizer role. The robot then moves to another area and continues searching. If no victims are found, the robot moves to another area and scans there. Once it has scanned its area, it sends the Find area to search task a complete message and terminates. Notice that concurrent tasks actually define a _plan_ on how to locate victims. The individual functions in the task are defined as functions on abstract data types or as low-level behaviors defined in the agent (robot) architecture.

![Figure 6. Locate Victim Task](image)

**Creating Agent Classes**

After each task is defined, we are ready for the phase. In the first step of the design phase, 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 may define a one-to-one mapping between roles and agent classes; however, we 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 roles are assigned to agent classes, the system organization is defined.

![Figure 7. Agent Class Diagram for Design 1](image)

The system shown in Figure 7 consists of only two types of agents: one playing the searching and organizing roles and one playing the searching and rescue roles (presumably based on the sensor/effecter packages on each robot). In
this case, since both the Search and Rescue agents can play the Searcher role, there are duplicate conversation types: 
victimFound, which is derived from the victim located protocol, and negotiate, which is derived from the negotiate areas protocol. The only other conversation is the findRescuer conversation, which is derived from the rescue auction protocol.

A different design, based on the same role model, is shown in Figure 8. In this design, we created a separate agent for the organizer role, which can reside on either a robot or a computer connected via a wireless network.

**Figure 8. Agent Class Diagram for Design 2**

**Constructing Conversations**

Once we have determined how to assign roles to agents, we can start Constructing Conversations, which is the next step in MaSE. A conversation defines a coordination protocol between exactly two agents and is modeled using two Communication Class Diagrams, one for the initiator and one for the responder. A Communication Class Diagram is a pair of finite state machines that define a conversation between two participant agent classes. Figure 9 shows the conversation extracted from the Locate Victim task for the Searcher and Organizer roles. Notice that this conversation will exist in our final system design regardless of the particular design we choose. The only difference between the two designs is the agents participating in the conversation, which is determined the agent playing the organizer role. To define conversations, we extract the communication between tasks.

**Figure 9. victimFound Conversation**

**Deployment Diagrams**

After defining the details of each conversation, the final design step is defining the implementation in its intended environment using Deployment Diagrams. In robotic applications, deployments define which agents are assigned to which robots. While in some cases, only one agent is allowed per robot; however, if sufficient processing power is available, there is not reason to limit the number of agents per robot. One possible deployment diagram for Design 1 is shown in Figure 10. In this case, there are two robots that have a rescue capability while only one has a search only capability. The lines between the agents denote communications channels and thus each agent may communicate directly with the others based on the allowable conversations.

**Figure 10. Deployment Diagram for Design 1**

A second deployment based on Design 2 is shown in Figure 11. In this case, we also have one searcher only and two rescuer robots. However, in this design, the Chief agent is separate from the Search agent. However, by putting the Chief agent on the same platform as the Search agent gives us basically the same design as Figure 10. In the case where we can only have one agent per platform, we could redesign the Agent Class Diagram and combine the appropriate roles into a single agent class.

**Figure 11. Deployment Diagram for Design 2**

A third alternative, also based on Design 2, is shown in Figure 12. In this deployment, all three robots have rescue agents. This shows how Design 2 is more adaptable to various configurations than Design 1 due to the separation of the searcher and organizer roles. Using Design 2, we are forced to have at least one Search only agent.

In each of these designs, the adaptability of the robot teams are somewhat limited by the fact that we have assigned only one robot to have the Chief agent, and thus only that robot is capable of playing the Organizer role. If that robot is lost or malfunctions, we have lost the ability of the team to function under this design. However, since the Organizer role is purely computational, there is no reason...
we cannot assign the Organizer role to each robot, thus providing redundancy. Deciding which role each robot should play then becomes a team decision. Reasoning about which roles to play is an area of future research.

![Deployment Diagram for Design 2](image)

**Figure 12. Deployment Diagram for Design 2**

**Implementation**

The platform for our search and rescue system is a set of Pioneer robots by ActivMedia running the Saphira interpreter and the Colbert programming language (ActivMedia 1997). An example of a system designed in MaSE using Saphira and Colbert is shown in Figure 13.

![MaSE Saphira/Colbert Architecture](image)

**Figure 13. MaSE Saphira/Colbert Architecture**

Tasks identified in the role model are implemented as Colbert activities, which run as separate threads in Saphira. Conversations are implemented as Colbert activities as well. Since tasks are assumed to execute concurrently in MaSE, the Saphira architecture provides a straightforward implementation environment.

Colbert does not allow complex data types; therefore, we must extend Colbert by adding data types defined using the MaSE domain modeling capability (currently under development). These data types are implemented via C functions that can be easily integrated into the Saphira environment. A global data area is required to pass data between activities since Colbert does not have the ability to return data from directly from activities.

To communicate with other robots, a message-oriented middleware (MOM) is being developed in C. The MOM package receives conversation requests and relays them to the agent via the Message Handler activity, whose job it is to start new conversations and task activities. After a conversation is started, each task or conversation can send or receive messages directly through MOM function calls.

**Conclusions**

From our initial investigation into designing robotic teams with MaSE, it appears that MaSE has the features required to design cooperative robot applications. Specifically, the concurrent tasks used during the analysis phase map nicely to the typical behaviors in robot architectures. MaSE also provides the high-level, top-down approach missing in many cooperative robot applications. While we only addressed high-level design here, MaSE uses a component-based approach to design the internal architecture of the agents, which can be applied to most any application. We also showed our proposed implementation architecture using the Saphira/Colbert environment.

Our future research plans include looking at fault tolerance and dynamic team reconfiguration based on the roles each robot is playing, or can play in the system. We also plan to provide a more detailed approach to mapping the high-level behaviors to low-level behaviors as defined in standard robotic architectures.

**References**


