Applying Agent Oriented Software Engineering to Cooperative Robotics

Scott A. DeLoach, Eric T. Matson, Yonghua Li

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 agent-oriented software engineering recently. However, many of these advances have not been applied to cooperative robotics even though earlier attempts at using agent approaches were successful (Drogoul and Collinot 1998). While a number of architectures have been developed, there have been few attempts at defining high-level approaches to cooperative robotics systems design (Parker 1998). In this paper, we attempt to determine the applicability of modern 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.
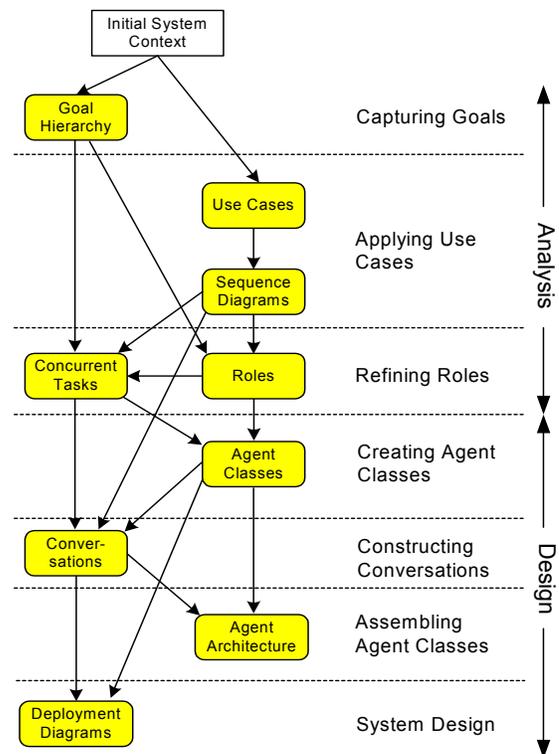

Figure 1. MaSE Methodology

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.

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.

Applying Use Cases

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

To help determine the actual communications required within a 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. The sequence diagram shows the events that occur when an agent playing the searcher role locates a victim. 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).
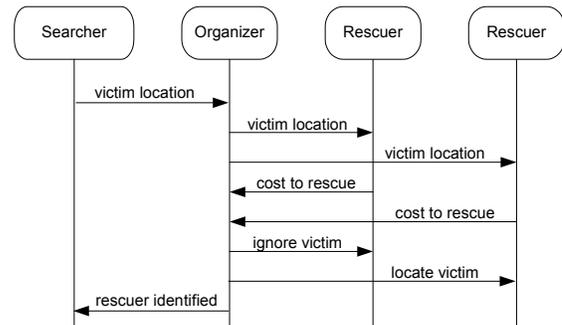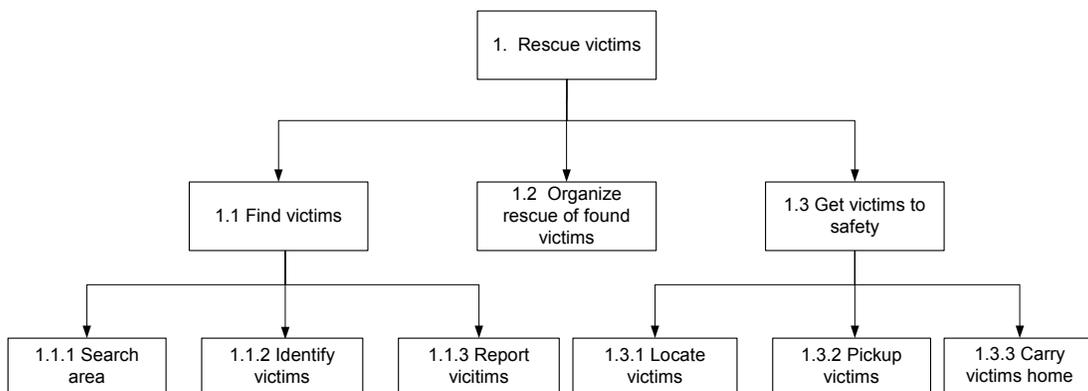
Figure 3. Sequence Diagram

Figure 2. Goal Hierarchy Diagram

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.
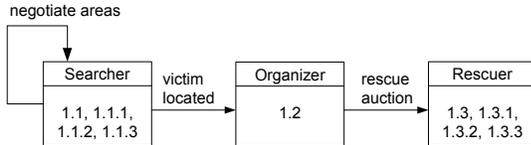

Figure 4. Role Model

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 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 step 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.

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 is means that if an event *trigger* is received with a number of arguments *args*1 and the condition *guard* holds, then the message *transmission* is sent with the set of arguments *args2* (all items are optional). Actions within each state are executed sequentially and are written as functions.
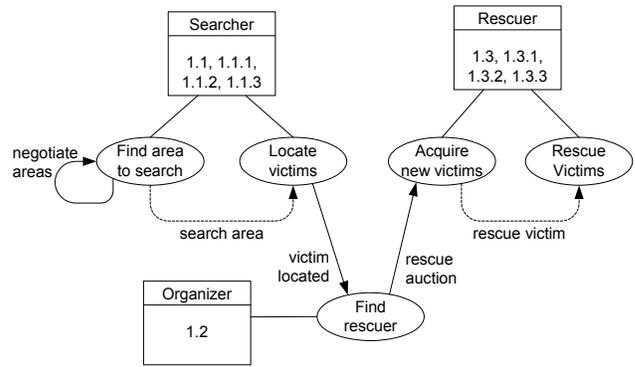

Figure 5. Role Model with Tasks

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.
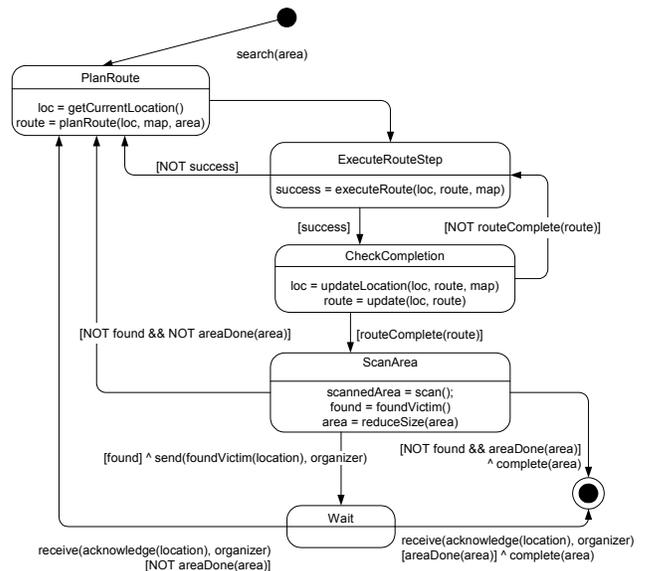

Figure 6. Locate Victim Task

Creating Agent Classes

After each task is defined, we are ready for the Design phase. In the first step, *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.
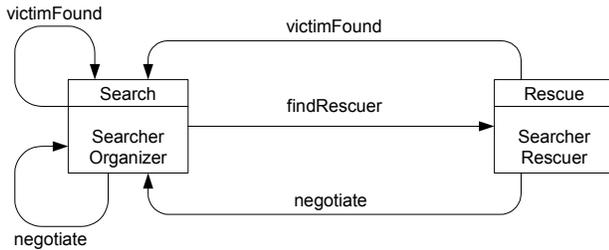

Figure 7. Agent Class Diagram for Design 1

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/effector 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 class for the organizer role, which can reside on a robot or on a computer connected via a wireless network.
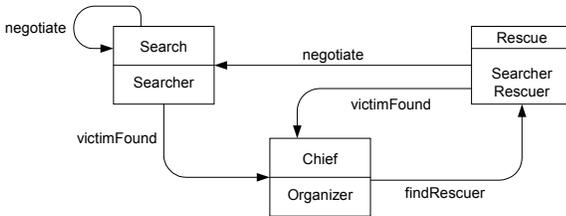

Figure 8. Agent Class Diagram for Design 2

Constructing Conversations

Once we have determined how to assign roles to agents, we can start *Constructing Conversations*. 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 which design we choose. The only difference between the two designs is the agents participating in the conversation, which is determined by who plays the organizer role.
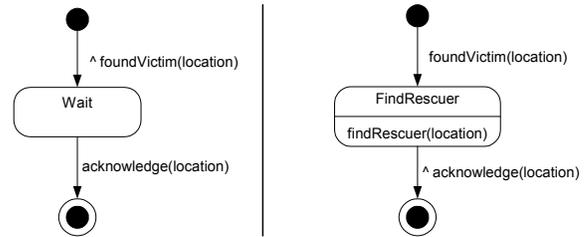

Figure 9. victimFound Conversation

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. In some cases, only one agent is allowed per robot; however, if sufficient processing power is available, there is no 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.
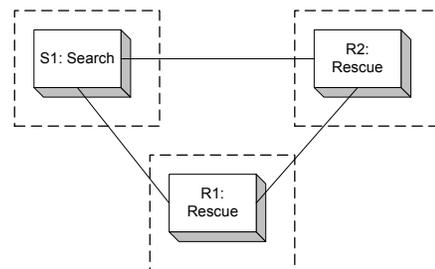

Figure 10. Deployment Diagram for Design 1

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. 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.

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.
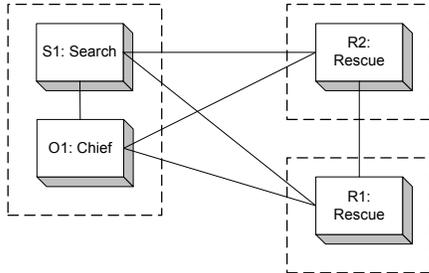

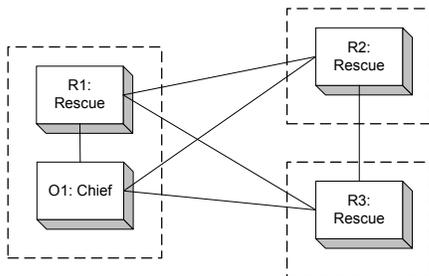**Figure 11. Deployment Diagram for Design 2**

Figure 12. Deployment Diagram for Design 2

In each design, team adaptivity is limited by our assignment of the *Chief* agent to a single robot, which means that only that robot can play the *Organizer* role. If that robot is lost or malfunctions, the team has lost its ability to function effectively. 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.

Implementation

Our original intention was to implement this design using a set of three Pioneer robots using the Colbert programming language under the Saphira interpreter (ActivMedia 1997) as shown in Figure 13. However, limitations of Colbert and Saphira required us to rethink our approach. Although MaSE tasks and conversations mapped nicely to Colbert activities, which run as separate threads, Colbert did not provide the required data types and data passing mechanisms required for an efficient implementation. Relying on Colbert and Saphira also limited our implementation to Pioneer robots instead of a heterogeneous mix of robots.

Therefore, we modified our initial approach and decided to use a heterogeneous set of robots. The platform for our search and rescue system was a group of two Nomad Scout and three Pioneer 2 robots (ActivMedia 1999). The resulting architecture of our system is shown in Figure 14, with an object model depiction of the same architecture shown in Figure 15. While we changed the underlying platform for the research, the analysis and design of the system did not change. The MaSE models developed for the original implementation were used almost without modification for the final system.
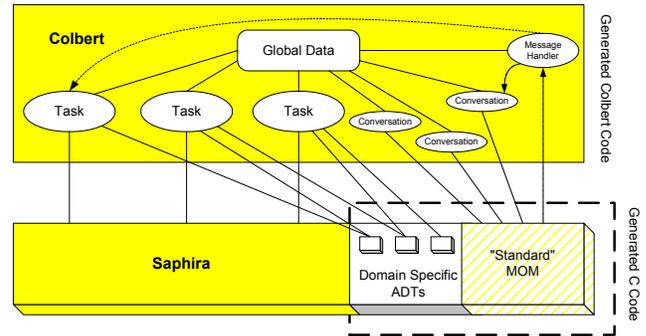

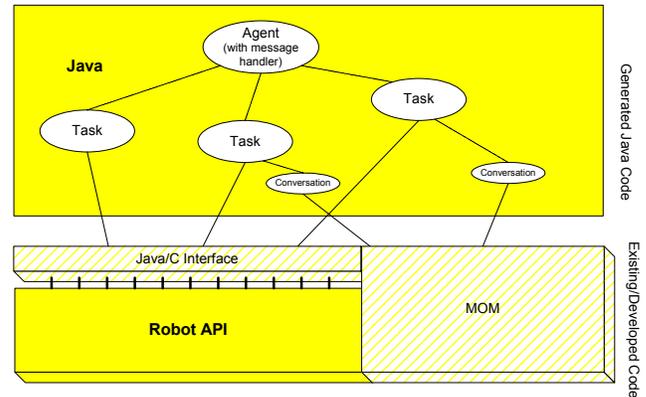**Figure 13. Implementation with Saphira/Colbert**

Figure 14. Search and Rescue Robot Architecture

In our approach, tasks identified in the role model are implemented as objects under control of the overall agent object. Conversations are implemented as Java objects and are associated with specific tasks. Since tasks are assumed to execute concurrently in MaSE, we use separate Java threads for each task. Conversations run in the same thread as their task to ensure the semantics of the MaSE task diagrams are preserved in the implementation.

To communicate with other robots, we developed a Java-based message-oriented middleware (MOM) that allows robots to send receive messages over designated sockets via wireless TCP/IP connections. The MOM package receives conversation requests and relays them to the agent via the Message Handler method in the agent, 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. All the software used to control the robots resides on laptops attached to each robot. We used two types of robots, those with rescue capability and those without. Figure 16 shows a Pioneer robot equipped with gripper capable of performing the rescuer role. (Here the victim is represented by a long cylindrical potato chip canister.)
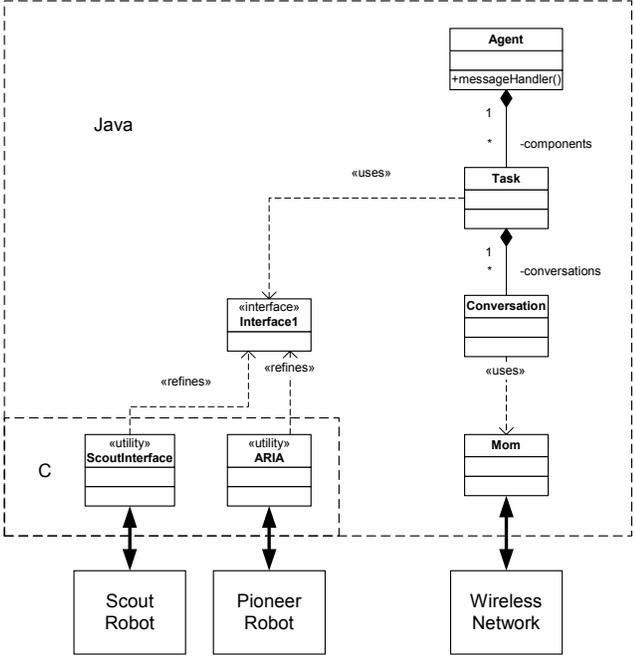

Figure 16. Pioneer Rescuing Victim

Figure 15. Object View of Architecture

Conclusions

From our initial investigation, 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 (we skipped the *Assembling Agent Classes* step), MaSE uses a component-based approach to designing the internal agent architectures, which is applicable in most any application. We also showed our proposed implementation architecture.

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

ActivMedia. 1997. Saphira Software Manual Version 6.1. ActivMedia Incorporated.

ActivMedia. 1999. Pioneer 2 Operating System Servers Operations Manual. ActivMedia Incorporated.

DeLoach, S. A., Wood, M. F., and Sparkman, C. H. 2001. Multiagent Systems Engineering, *The International Journal of Software Engineering and Knowledge Engineering* 11(3):231-258.

Drogoul, A., and Collinot A. 1998. Applying an Agent-Oriented Methodology to the Design of Artificial Organisations: a case study in robotic soccer. *Autonomous Agents and Multi-Agent Systems*, 1(1): 113-129.

Murphy, A. L., Pietro Picco, G., and Roman, G. C. 2001. LIME: A Middleware for Physical and Logical Mobility. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21)*. 524-233.

Parker, L. E. 1996. On the Design of Behavior-Based Multi-Robot Teams. *Advanced Robotics*, 10(6):547-578.

Parker, L. E. 1998. Toward the Automated Synthesis of Cooperative Mobile Robot Teams, *Proceedings of SPIE Mobile Robots XIII,* Volume 3525:82-93.

Zambonelli, F., Jennings, N. R., and Wooldridge, M. J. Organisational Rules as an Abstraction for the Analysis and Design of Multi-Agent Systems. *International Journal of Software Engineering and Knowledge Engineering.* 11(3): 303-328.