International Journal of Pattern Recognition and Artificial Intelligence

EXPLOITING AGENT ORIENTED SOFTWARE ENGINEERING IN COOPERATIVE ROBOTICS SEARCH AND RESCUE

SCOTT A. DELOACH, ERIC T. MATSON, YONGHUA LI

Department of Computing and Information Sciences, Kansas State University
234 Nichols Hall, Manhattan, Kansas, United States of America 66506
{*sdeloach, etm7766, yli3568*}*@cis.ksu.edu*

This paper reports our progress in applying multiagent systems analysis and design in the area of cooperative robotics. 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 cooperative robotic 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.

Keywords: Cooperative robotics; multiagent systems; software engineering

1. Introduction

There have been many recent advances in agent-oriented software engineering. Unfortunately, many of these advances have not carried over to cooperative robotics even though earlier attempts at using agent approaches were successful [6]. While many architectures have been developed, there have been few attempts at defining high-level approaches to cooperative robotics systems design [12]. 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, adaptivity, and fault tolerance [11].

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 cooperative robotic systems instead of the behavior-based, bottom up approach employed in most robotic implementations [3,11,13]. Other MAS methodologies were considered, but were rejected for reasons described in Section 6. 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.

2. Designing Cooperative Robotic Systems

We chose to use the MaSE methodology [4] to design our cooperative robotic system because it provides a top-down approach and a detailed sequence of models for developing multiagent systems. The seven-step MaSE process and its associated models are as follows:

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 goal of MaSE is to guide a system developer from an initial system specification through implementation using a set of inter-related system models.

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 [11].

2.1. *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 1). 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 the sub-goals that are necessary to satisfy parent goals. Eventually, each goal will be associated with roles and agent classes that are responsible for satisfying that goal.

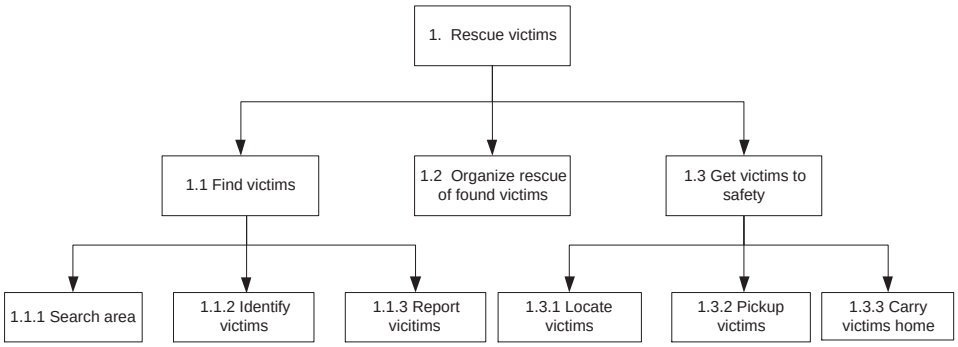


Fig. 1. Goal Hierarchy Diagram

Figure 1 shows the initial high-level goal hierarchy for the robotic search and rescue domain. Obviously, this could be further decomposed; however, the purpose of the goal hierarchy diagram in MaSE is to identify the main system level goals, not individual agent goals.

2.2. *Applying Use Cases*

Applying Use Cases 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 2. 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 Figure 2, 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.) and the organizer selects the most appropriate rescuer. The organizer notifies the rescuers of the selection 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

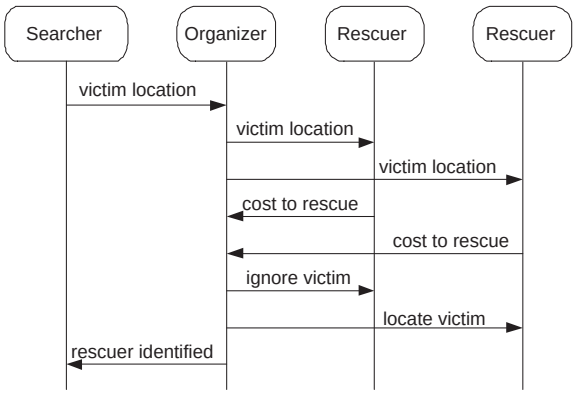


Fig. 2. Sequence Diagram

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 [9].

2.3. *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. Role definitions are captured in a standard Role Model as shown in Figure 3. The numbers in the role boxes show the allocation of goals (as numbered in Figure 1) to the individual roles.

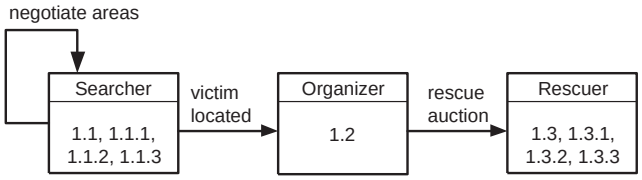


Fig. 3. Role Model

In our search and rescue system, we identified three roles: searcher, rescuer, and organizer. The role model can be used to explain high-level system operation.

In Figure 3 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. We chose a distributed approach to controlling the search while using a centralized approach to controlling the rescue (via the Organizer role) to show the applicability of MaSE to both control structures.

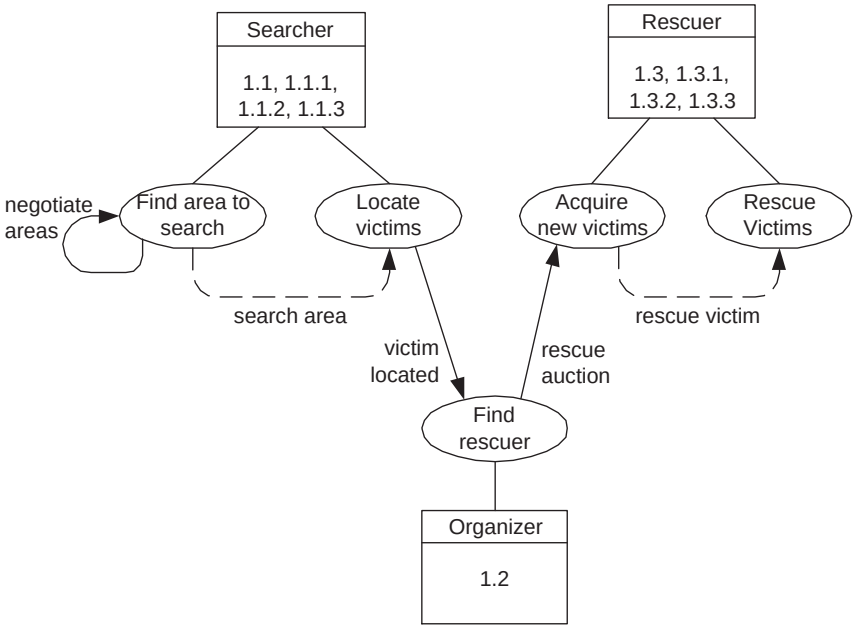


Fig. 4. Role Model with Tasks

2.3.1. *Defining Tasks*

Once roles have been identified, the detailed tasks, which define how a role accomplishes its goals, are defined and assigned to specific roles. A set of concurrent tasks provides 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 4. The ellipses denote tasks performed by the attached role while the arrows between tasks define protocols that specify how communication is performed. 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 5. The syntax of state transitions is *trigger(args1) [guard]* $\wedge$ *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* (each part of the syntax is optional as null transitions are allowed). Actions within each state are executed sequentially and are written as functions. The && symbol is the logical *and* operator.

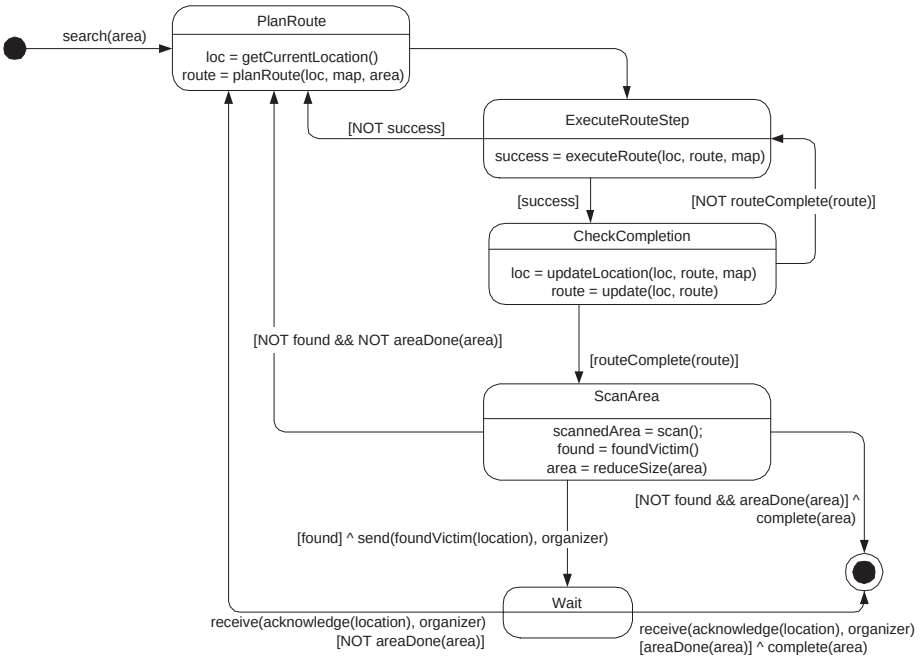


Fig. 5. Locate Victim Task

Locate Victim 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 (in the PlanRoute state) and then goes about executing the route in the ExecuteRouteState. If route execution fails (in CheckCompletion), the task re-plans the route and updates its location on its local map. When the robot gets to its area,

it scans the area for victims in the ScanArea state. If a victim is found, it notifies an *organizer* role and enters the Wait state to wait for acknowledgement. Once it has received acknowledgement, the robot 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.

2.4. *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 6. 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.

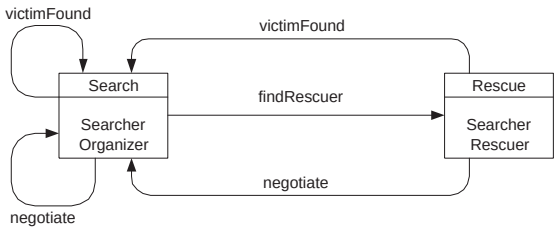


Fig. 6. Agent Class Diagram for Design 1

The system shown in Figure 6 consists of only two types of agents: one playing the searcher and organizer roles and one playing the searcher and rescuer 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 7. 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. We also introduce an explicit *Chief* agent who is responsible for the Organizer role.

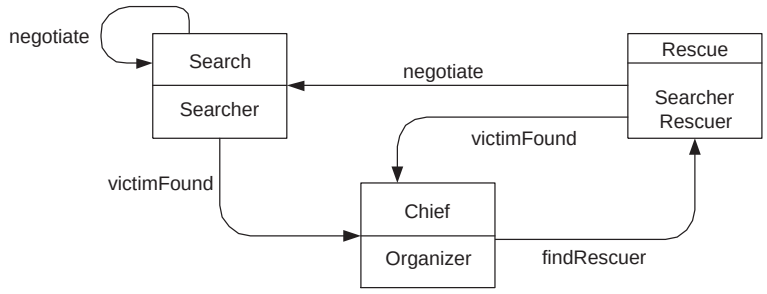


Fig. 7. Agent Class Diagram for Design 2

2.5. *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 and uses the same syntax as the Task diagrams. Figure 8 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.

2.6. *Deployment Diagrams*

After defining 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 9. In this case, two robots have a rescue capability while the third robot 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.

A second deployment based on Design 2 is shown in Figure 10. In this case, we also have only one searcher and two rescuers, but the *Chief* agent is separate

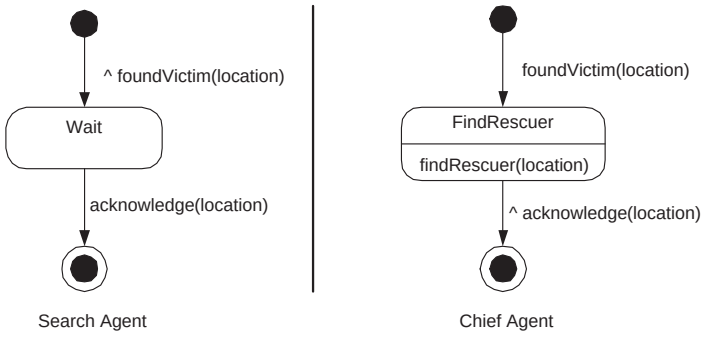


Fig. 8. victimFound Conversation

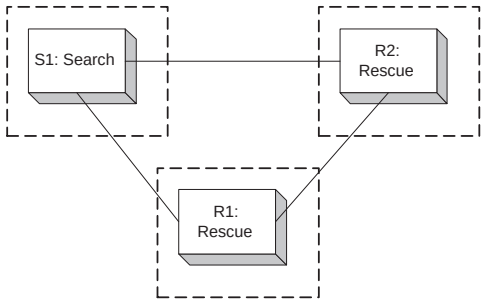


Fig. 9. Deployment Diagram for Design 1

from the *Search* agent. Putting the Chief agent on the same platform as the Search agent gives us the same design as Figure 9. 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 11. 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 1, we are forced to have at least one Search only agent.

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 an *Organizer* role to each robot, thus providing redundancy. Deciding

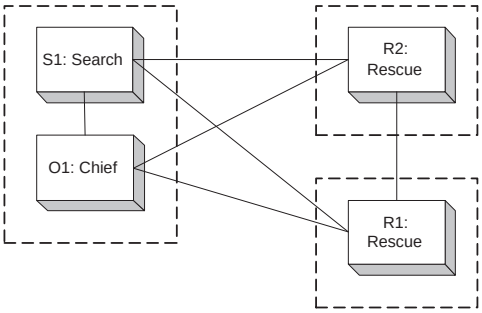


Fig. 10. Deployment Diagram for Design 2

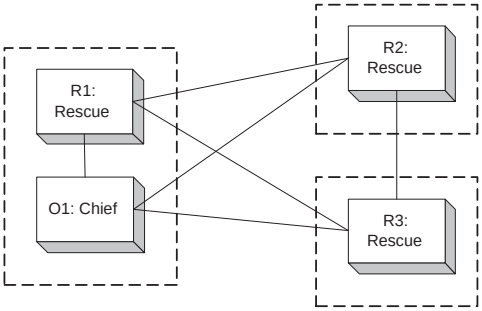


Fig. 11. Deployment Diagram for Design 2

which role each robot should play then becomes a team decision. Reasoning about which roles to play is an area of future research.

3. 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 [1]. However, limitations of Colbert and Saphira and our desire to use a heterogeneous set of robots 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 truly heterogeneous mix of robots.

Therefore, we modified our initial approach and used a heterogeneous set of robots that included a group of two Nomad Scout and one Pioneer 2 robots [2]. The resulting robot architecture is shown in Figure 12, which is a standard UML class

diagram. The architecture has a single *Agent* object that controls the overall execution of the agents. Each agent has a set of *Task* objects that directly implement the tasks defined in the design. Each task has a set of *Conversation* objects that implement the required conversations via our messaging package (the *Mom* object). Each task object actually sends commands to the robot via an *interface* object, which provides a *wrapper* that encapsulates the individual robot control software. 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 without modification in the final implementation.

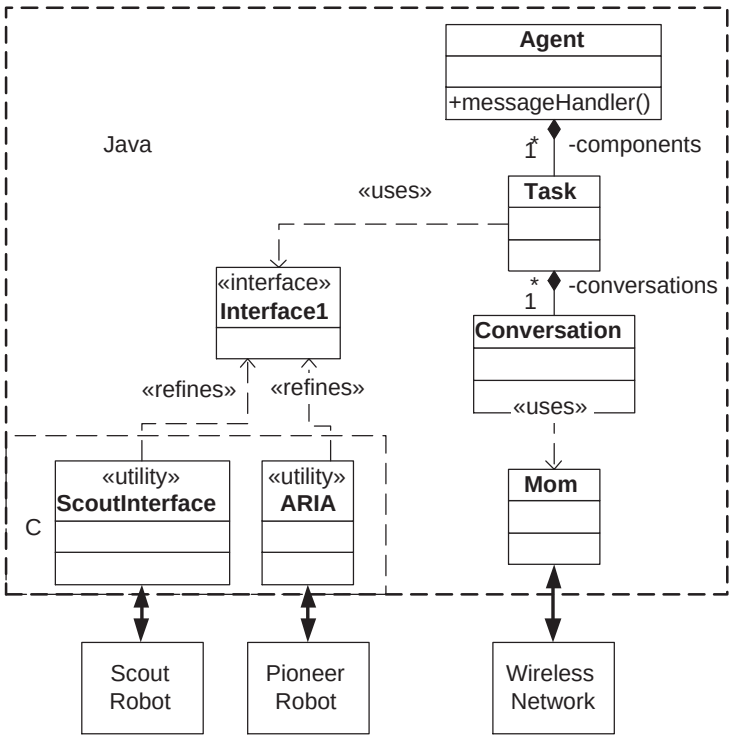


Fig. 12. Object View of Architecture

As described above, tasks identified in the role model are implemented as objects under control of a single 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 that the semantics of the MaSE task diagrams are preserved in the implementation.

To communicate with other robots, we used a Java-based message-oriented middleware (MOM) that allows robots to send and 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 resided on laptops attached to each robot. We used two types of robots, those with rescue capability and those without. Figure 13 shows one of our Pioneer robots equipped with gripper capable of performing the rescuer role. (Here the victim is represented by a long cylindrical potato chip canister.)

Fig. 13. Pioneer Rescuing Victim

4. Experimental Results

We used Design 2 to implement our experimental team. Due to the small area in which we had to conduct the experimental runs, we limited the team size to three robots: two searchers and one rescuer. The Chief agent was deployed alternately on a rescue robot as well as on a separate computer. We used two basic configurations. In one set of experiments, we used a single searcher and a single rescuer while in the second set of experiments we used the entire team, two searchers and one rescuer. We did not add additional robots to the team due the size limitations of our testbed (approximately 9 meters square).

The test area for a single searcher, single rescuer scenario was set up as shown in Figure 14(a). The robots were given the nominal size of the search area and they assumed there were no obstacles in that area other than victims. Both robots

were started at approximately the same time. The searcher started its search and used the search pattern shown by the dotted lines. The rescuer (the robot with the square on its front) waited at the "home base" for a message from the searcher with the coordinates of a possible victim. When the searcher found an object in the room that appeared to be the approximate size of a victim, it started the victimFound conversation with the Chief agent (which was running on either on the rescuer robot or a separate computer). The Chief agent then sent messages to all known rescuers (in this case only one), waited for them to respond with their cost (we used a simple linear distance calculation from the rescuers current position to the victim including in intervening rescues that had been cued up), and then selected the lowest cost rescuer. While we did not run any physical experiments with more than one rescuer, we did simulate multiple rescuers to show that this protocol did work as intended. Once a rescuer was selected, the Chief initiated a findRescuer conversation with the rescuer, which started the rescue robot on its rescue mission.

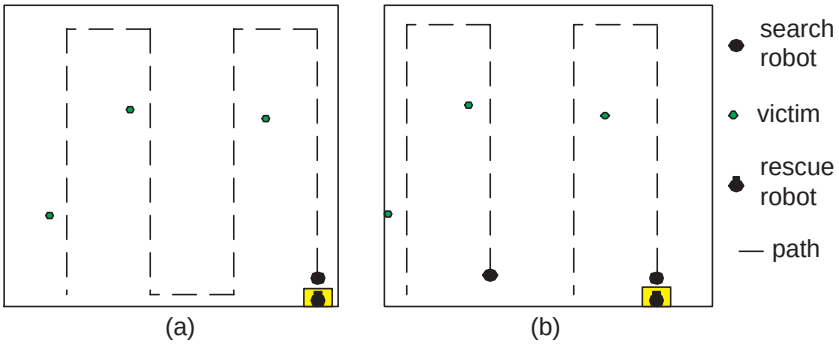


Fig. 14. Experiment Layouts (1 and 2 searcher)

Once the rescuer received permission to rescue the victim from the Chief, it immediately set out in a straight line to rescue the victim. If it was currently in the process of rescuing another victim, the rescuer put the victim's coordinates into a rescue queue. When it completed the first rescue, the rescuer checked its queue to see if it had any other victims to rescue. The rescue robot proceeded directly to the reported location of the victim and then began a local search for the victim. (Coordinates passed to the robot were often slightly off due to wheel slippage, etc.) The robot searched a one-meter radius for the victim. Upon locating the victim, the robot aligned itself and moved toward the victim in order to pick it up with its grippers. After the victim was picked up, the rescuer returned to home base and set the victim down. Once the victim was set down, the rescuer checked its queue to see if any more victims needed rescuing. Due to the small search area,

whenever there was more than one victim, the searcher generally completed its search before the rescuer had completed its first rescue. Thus, the queue of victims was used extensively. If the rescuer could not find a victim in the one-meter radius, it assumed that it had received an erroneous report and abandoned the search; it then immediately checked its queue to see if any more rescues were needed. If not, it returned to home base.

4.1. *Data*

In all, we performed 55 rescue scenarios using the team consisting of one searcher and one rescuer. Of these, 27 had single victims, 19 had two victims, and 9 had three victims. All 27 of the single victim scenarios were successful, with the rescuer finding and rescuing the victim. In addition, since there were no spurious "ghost" [a] sightings, the number of messages sent in each experiment was six, the number expected given the design. The results using time taken versus the distance to victim are shown in Figure 15(a). Generally these are fairly linear (as one would expect) with the variance being explained by the amount of time the rescuer had to "search" for the victim within the one meter area once it got to the reported location. The fact that the variance increases as the distance increases is explained by the fact that the further the searcher and rescuer travel, the more error is introduced into their estimate of their own position. (None of the robots had any sort of internal navigation system or compass to help them.)

When we went to two victims (still with a single searcher and rescuer), we experienced 5 failures in the 19 experiments (38 victims) for a success rate of 87%. In five of the two victim experiments, the searcher picked up "ghost" readings of victims due to its close proximity to the rescuer. In all, there were seven ghost victims reported: a single ghost in three experiments and two ghosts apiece in the other two. For each ghost reported, an extra six messages were sent and the rescuer had to spend extra time searching for a victim that did not exist. Furthermore, in one experiment, the searcher failed to find one of the victims and thus reported only a single victim. This put the average number of messages per experiment to 13.9 instead of the nominal 12. Of the 14 completely successful experiments, the time versus distance comparisons have a similar graph to those for the single victim (see Figure 15(b)).

The three-victim case for a single searcher/single rescuer configuration developed even more problems of the type experienced in the two-victim case. Out of nine runs, there were five "ghosts" reported, each of which can be attributed to direct sensing of the rescuer by the searcher or to the searcher actually picking up the sonar "pings" of the rescuer while it was trying to find a reported victim. This caused the system to generate approximately 21.3 messages per run versus

[a] We defined a "ghost" as an invalid identification of a rescue victim. Most often, these were other robots or inadvertent sonar pings from the other robobts.

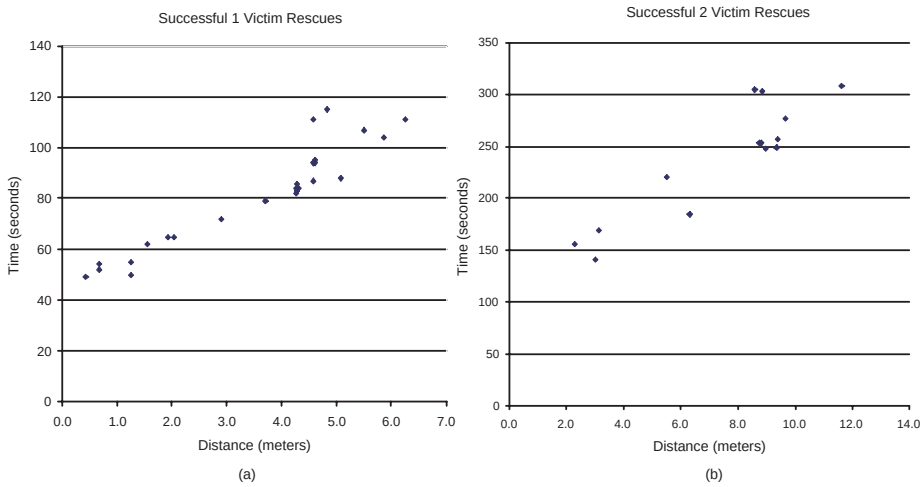


Fig. 15. Single and Two Victim (one searcher, one rescuer)

the optimal 18. Overall, the number of victims rescued was 21 out of 27 for 78%. Most of the failures were due to the rescuer failing to find or grasp the victims. This was due to the errors introduced by each successive start/stop/turn of the robot. The more victims the rescuer had to rescue, the greater the error grew. This was evidenced by the rescuer never failing to find the first victim, while finding only five of nine third victims.

The final test scenario consisted of a team of two searcher robots and a single rescuer as shown in Figure 14(b). In this case, the number of robots for the size room was at the maximum. Because each of the robots was using the same basic sonar device, the number of "ghosts" from the robots picking up each other's sonar pings grew significantly and made testing troublesome. The two searcher robots were both Scouts, while the rescuer was a Pioneer. In this case, the searchers started at various locations (with each knowing its starting location) and searched their assigned parts of the room. The rescuer (via the Chief agent) responded to either of the searchers on a first come, first served basis. The ghosting problems, which were exaggerated by the introduction of another robot, caused there to be four ghosts in the eight successful runs of the system. During these eight runs, the success rate was 62% (16 out of 24 victims), while the average number of messages per run was 21 (versus the optimal 18). Only in one case did the system rescue all three victims.

5. Methodology Evaluation

After completing the experiments, we attempted to evaluate the MaSE methodology and its usefulness in developing cooperative robotic applications. Our general conclusion is that the MaSE methodology worked very well and was not responsible for the problems encountered during system execution. All of the problems stemmed from low-level sensing and robot control problems; we had no problems related to overall system design or communications between robots. The remainder of this section discusses the applicability of each step in MaSE.

5.1. *Analysis Phase*

As described above, the MaSE analysis phase consists of Capturing Goals, Applying Use Cases, Refining Roles, and Defining Tasks. The first step, Capturing Goals, proved to be very useful and made the rest of the analysis and design much simpler. The key goal turned out to be goal 1.2, Organize rescue of found victims. Identifying the organization of the rescue as a high-level goal actually drove the creation of the Organizer role (and eventually the Chief agent), which was responsible for much of the actual teamwork achieved during the rescues. The Applying Use Cases step was also critical to the success of our eventual design as it allowed us to step through how the team should operate long before getting into design. It was during this step that the roles were and most of the required interactions were identified, which lead naturally to the definition of the role model in the Refining Roles step. Explicitly assigning goals and sub-goals to individual roles made role responsibilities clear and made the identification of tasks for each role straightforward. Once the tasks were identified, the use cases defined previously were of great help in determining inter-task communication paths.

Once the tasks and inter-task communications paths were identified, developing the task diagrams was also straightforward, although their complete development took more time than the previous steps and required multiple iterations. However, once again the use cases were very helpful in creating the initial flow of control within the tasks. Generally, the first iteration of a task was developed by looking at the sequence of messages sent or received in the various use cases and putting states between each message. We then determined the processing that had to be accomplished to send the appropriate messages and provide appropriate control. Defining the actual message data also helped us determine the exact processing requirements in each state.

Overall, the analysis phase went very smoothly and followed the prescribed MaSE process. Goal identification was key, as it drove the number and types of use cases we eventually created. The use cases were critical in defining the roles, tasks, and inter-task communications.

5.2. *Design and Implementation*

After completing the analysis phase, the system design was a relatively simple process. Once we decided which agents would be assigned to play which roles (we used Design 2 as shown in Figure 7), we used the agentTool system to support our design. The agentTool[b] system is a graphical modeling tool that supports the MaSE methodology and provides automatic analysis to design transformations as well as some code generation capability [5]. To complete the system design, we used agentTool's semi-automated design transformations. We also used the agentTool code generation capability to generate the initial robot shells and the inter-robot conversation code, which contributed to the lack of problems in team interaction.

Using MaSE also allowed us to consider the effects of proposed changes to deal with some of the problems we encountered. For instance, we determined that one reason for the reporting of ghost victims was the fact that goal 1.1.2, Identify Victims, was not being performed adequately. The searcher was assuming that any object it picked up was a victim. One solution we considered was to make the searcher's Locate Victims task more robust, or to add a separate Identify Victim task. Both of these solutions required only changes to the searcher agent.

A second approach to solving the ghost problem was to add a new role (and separate agent/robot) to ensure there was a victim present at a reported location before sending in the rescuer. Using the analysis phase artifacts, we determined that this change would only require a change to the Organizer role (Chief agent), which would be responsible for finding an "identifier" robot and verifying the location of the victim before actually tasking a rescuer.

The third approach proposed to alleviate the ghost problem was to have each robot continually inform every searcher robot of its current location so that the searcher could "rule out" possible victims as other robots. Again, analysis of the MaSE models showed that this proposal would require additional tasks in both the searcher and rescuer roles to broadcast its location. It would also require the searcher role to be able to receive such messages and integrate them into its Locate Victim task. Ultimately, we did not implement any of these proposed changes since none of them could totally alleviate the problems we were encountering. However, the existence of the MaSE models made the analysis of each proposal straightforward and easy to justify. Thus, not only did we find MaSE useful in the analysis and design of our cooperative robotic system, but we also believe that it would be of great benefit in the maintenance and modification of such systems as well.

6. Conclusions

From this initial investigation, it appears that MaSE has the features required to be of great benefit in the analysis and design cooperative robot applications. MaSE provides the high-level, top-down approach missing in many cooperative

[b] agentTool is freely available at http://www.cis.ksu.edu/∼sdeloach/ai/agentool.htm

robot applications. Moreover, the existence of the MaSE analysis and design models is also helpful in the maintenance and modification of cooperative robotic systems. Although not specifically addressed in this design, concurrent tasks used during the analysis phase map nicely to the typical types of low-level behaviors found in behavior-based robotic architectures.

While MaSE appears to have the required features to make it a good choice for developing cooperative robotic applications, this does not imply that all multiagent methodologies are equally applicable. For instance, the GAIA methodology [14], while providing good analysis coverage of individual agent plans and actions, falls short when defining the interactions between agents. Likewise, Tropos [7] provides extensive support for early requirements analysis, but does not provide abstract models that can easily represent reactive tasks or inter-agent communications protocols. Other methods, such as the AAII method[8] focus on particular agent architectures that may not be conducive towards heterogeneous cooperative robotic applications. While not a methodology per se, the Agent UML (Unified Modeling Language) [10] is an effort to extend UML to provide the tools to model concepts of interest to multiagent systems. These models allow cooperative robotic developers to use state-of-the-art UML models augmented with special notations for modeling communication and autonomous goal based behavior important to cooperative robotic applications.

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

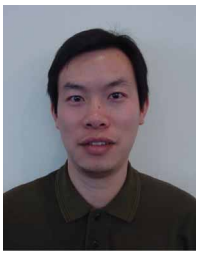
1. ActivMedia. 1997. *Saphira Software Manual Version 6.1.* ActivMedia Incorporated.
2. ActivMedia. 1999. Pioneer 2 Operating System Servers Operations Manual. ActivMedia Incorporated.
3. Arkin, R.C., Balch, T. 1997. AuRA: Principles and Practice in Review. *Journal of Experimental and Theoretical Artificial Intelligence* 9(2-3):175 - 188.
4. 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.
5. DeLoach, S. A., and Wood, M. F. 2001. Developing Multiagent Systems with agentTool. in *Intelligent Agents VII. Agent Theories Architectures and Languages, 7th International Workshop* (ATAL 2000, Boston, MA, USA, July 7-9, 2000), C. Castelfranchi, Y. Lesperance (Eds.). Lecture Notes in Computer Science. Vol. 1986, Springer Verlag, Berlin.
6. 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.
7. Giunchiglia, F. Mylopoulos, J. and Perini, A. 2002. The Tropos Software Development Methodology: Processes, Models and Diagrams. *The Third International Workshop on Agent-Oriented Software Engineering (AOSE-2002).* Bologna, Italy. July 2002.

8. Kinny, D. and Georgeff, M. 1996. *A Methodology and Modelling Technique for Systems of BDI Agents*. Technical Note 58. Australian Institute for AI. January 1996.
9. 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.
10. Odell, J., Parunak, H.V.D and Bauer, B. 2001. Representing Agent Interaction Protocols in UML, *Agent-Oriented Software Engineering*, Paolo Ciancarini and Michael Wooldridge eds., Springer, Berlin, pp. 121-140.
11. Parker, L. E. 1996. On the Design of Behavior-Based Multi-Robot Teams. *Advanced Robotics*, 10(6):547-578.
12. Parker, L. E. 1998. Toward the Automated Synthesis of Cooperative Mobile Robot Teams, *Proceedings of SPIE Mobile Robots XIII*, Volume 3525:82-93.
13. Parker, L. E. 2000. Current state of the art in distributed autonomous mobile robotics. In *Proceedings of the International Symposium on Distributed Autonomous Robotic Systems*, pp. 3-12.
14. Wooldridge, M., Jennings, N., and Kinny, D. 2000. The Gaia Methodology for Agent-Oriented Analysis and Design, *Journal of Autonomous Agents and Multi-Agent Systems*. vol. 3(3).

Biographical Sketch and Photo

Scott DeLoach received his B.S. in computer engineering from Iowa State University in 1982 and his M.S. and Ph.D. in computer engineering from the Air Force Institute of Technology in 1987 and 1996 respectively. He is currently an assistant professor with Kansas State University.

Eric Matson is currently a Ph.D. student in the Department of Computing and Information Sciences, College of Engineering at Kansas State University. Eric received his BS in Computer Science from Kansas State University in 1988, MBA in Operations Management from The Ohio State University in 1993, and MSE in Software Engineering from Kansas State University in 2002.



Yonghua Li received the B.S. in chemistry from Lanzhou University in 1991, an M.S. in polymer material engineering from Beijing University of Chemical Technology in 1996, and an master of software engineering from Kansas State University in 2002. Currently, he is a PhD student in computer science, Kansas State University.