GIT-COGSCI-95/02

Specification and Execution of Multiagent Missions

Douglas C. MacKenzie
Jonathan M. Cameron
Ronald C. Arkin

January 24, 1995

Douglas C. MacKenzie
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332 U.S.A.
doug@cc.gatech.edu

Abstract

Specifying a purely reactive behavioral configuration for use by a multiagent team executing a mission requires both a careful choice of the behavior set and the creation of a temporal chain of behaviors which executes the mission. This difficult task is simplified by applying an object-oriented approach to the design of sequences of behavioral configurations where a methodology called temporal sequencing is used to partition the mission into discrete operating states and enumerate the perceptual triggers which cause transitions between those states. Several smaller independent configurations can then be created with each implementing one state, completing one step in the sequence. When properly constructed, these configurations (assemblages) become high level primitives reusable in subsequent projects, reducing development time.

In the multi-vehicle domain being studied for the ARPA Demo II project, assemblages such as travel_to_location and occupy_location consist of groups of basic behaviors associated with coordination mechanisms that allow the group to be treated as a single, coherent behavior. For example, travel_to_location consists of **move_to_goal**, **avoid_obstacle**, **avoid_robot**, **noise**, *and* **stay_in_formation** *primitive behaviors moderated by a cooperative coordination operator. Upon instantiation, the assemblage is parameterized with a particular formation, goal location, and termination conditions. A mission coordination operator determines which assemblage to activate based upon the mission being executed and the current state of the system.*

A scenario language has been developed which allows specifying missions as sequences of steps, where each step invokes a particular assemblage. The missions are specified in a structured user-friendly language targeted for groups of cooperating robotic vehicles executing military-style scout missions. Various multiagent missions have been demonstrated in simulation using this system. Deployment on Denning mobile robots demonstrates the utility of this mission execution system, while later deployment on the ARPA Demo II test platforms will ultimately allow comparisons with software developed using other methods.

1 Introduction

Reactive behavior-based architectures[1, 7] decompose a robot's control program into a collection of behaviors and coordination mechanisms. The overt, visible behavior of the robot arises from the emergent interactions of these behaviors. The decomposition process further allows for the construction of a library of reusable behaviors by designers skilled in low-level control issues. Subsequent developers using these components need only be concerned with their specified functionality. Further abstraction can be achieved by permitting construction of assemblages from these low-level behaviors which embody the abilities required to exhibit a complex skill.

Creating a multiagent robot configuration involves three steps; determining an appropriate set of skills for each of the vehicles; translating those mission-oriented skills into sets of suitable behaviors (assemblages); and the construction/selection of suitable coordination mechanisms to ensure that the correct skill assemblages are deployed correctly over the temporal sequence of the mission. The mission specification is facilitated by the creation of an interpreter capable of activating and parameterizing skill assemblages from commands given in a high-level mission specification language. This allows the robot commander to design the mission using a structured language either on-line or via a mission description file. In either case, the commander is presented with high-level commands, such as move_to_location, and need not know anything about robot control programming. This supports use by personnel with minimal system training but who are highly skilled in the domain tasks the robots are ordered to perform.

The construction and functionality of the Georgia Tech *MissionLab* software environment that is based upon this philosophy is documented in this paper. Several different levels of competence are provided for access into the system. The primitive behavior implementor must be familiar with the particular robot architecture in use and a suitable programming language such as C++. However, at a higher level, using a library of behaviors to construct skill assemblages does not require programming knowledge since a graphical editor has been developed which allows visual placement and connection of behaviors. The construction of useful assemblages, however, still requires knowledge of behavior-based robot control. At the highest level, specifying a configuration for the robot team consists of selecting which of the available skills are useful for the targeted environments and missions. This process can also be completed graphically. Using the mission coordination module, specification of actual missions can occur at run-time using a domain-specific structured language. Reflecting the targeting of this research for the ARPA UGV community[8], military terminology and nomenclature are currently used in *MissionLab* to facilitate specification of missions by military users unfamiliar with robot control techniques. The overall philosophy, however, is by no means restricted to this application domain.

After a review of related work in Section 2, Section 3 presents specification methods for primitive behaviors, skill assemblages, and configurations. The *MissionLab* system is presented in Section 4.1 and the mission scenario language is documented in Section 4.2. along with simulation results. Supporting runs using Denning robots substantiating the simulation results are presented in Section 5. The summary and conclusions given in Section 6 complete the paper.

2 Related Work

The on-going research described here draws heavily on research conducted by others in the field. Appropriate comparisons to related research will allow the reader to understand how this research is positioned relative to other existing work.

Developed at CMU, the mission specification language, SAUSAGES[9, 10], allows specification of a robot mission as a sequence of operating states and a collection of state transitions, similar to the capabilities of the mission coordination operator and mission scenario language. However, the flat graph-like structure of SAUSAGES does not provide support for abstraction. Since SAUSAGES is used in the ARPA UGV program, a SAUSAGES code generator will be developed within *MissionLab* to allow targeting the UGV architecture.

Lyons' Robot Schemas (RS)[16] architecture is based on the port automata model using synchronous communication. RS introduced the notion of a coordinated assemblage of components which is treated as a new component. RS-L3[15] is a discrete event systems variant of RS which has been used to implement and analyze a robotic work cell. The specification of robot configurations presented in Section 3 implements several of the ideas first presented in RS.

Multivalued logic has been used as a mechanism for the analysis of coordinated assemblages[19]. Analysis of the correctness of configurations is necessary to support the creation of configurations by novice users. It is intended that support for such analysis will be developed as part of this research. Multivalued logic techniques are expected to prove useful in such analysis.

The REX/Gapps architecture[12] supports situated formal analysis by constructing the control program in the form of a synchronous digital circuit. Analysis, however, requires a detailed environmental model which is unreasonable to expect to exist in all but highly structured environments.

Our Graphic Designer's support for simple construction of assemblages draws heavily on experience with the Khoros[13] image processing workbench. Khoros allows the user to select items from a library of procedures and place them on the work area as glyphs. Connecting dataflows between the glyphs completes construction of the "program". Each glyph in Khoros represents a UNIX program which is instantiated as a separate UNIX process.

3 Configuration Specification

Configuration design tools have been developed which support the graphical construction of abstract configurations which are both robot and architecture independent. Figure 1 shows the architecture diagram for the configuration development system. The Graphic Designer is used to create and maintain configurations specified in the Configuration Description Language (CDL). CDL supports the recursive construction of reusable components at all levels, from primitive motor behaviors to societies of cooperating robots. The Graphic Designer supports this recursive nature by allowing creation of coordinated assemblages of components which are then treated as atomic higher-level components available for later reuse. The CDL compiler generates intermediate code in the Configuration Network Language (CNL) to minimize the complexity of the CDL compiler and allow incremental development of the design tools. The architecture and robot binding process determines which CNL compiler will be used to generate the final executable code, as well as which libraries of behavior primitives will be used. A schema-based C++ CNL compiler has been developed and a SAUSAGES CNL compiler targeting the ARPA UGV architecture is planned. The compiled executables will either drive the targeted vehicles or a suitable simulation. The Georgia Tech *MissionLab* simulation and operator console is described in Section 4.1.

3.1 The Configuration Description Language (CDL)

The context-free Configuration Description Language (CDL) provides a solid theoretical foundation for specifying architecture and robot independent configurations for societies of behavior-based robots. The language specifies the coordination between members of homogeneous teams, of heterogeneous castes, assemblages of behaviors on individual robots, as well as perceptual strategies within primitive sensorimotor behaviors.

The grammar G generating the language is described by the notation[11] $G = (V, T, Q, S)$, where V is the set of variables, T is the set of terminal symbols, Q is the set of productions, and S represents the highest-level society (the start variable). Using this notation, G is described as:

$$\begin{aligned} G \;=\; & (\{S, X, R, A, C, Y, B, P, Z\}, \\ & \{\mathbf{p}_i, \mathbf{m}_j, \mathbf{a}_k, /_l, *_m, +_n, -_o, \%_p, @_q, \#_r, =_s, \{, \}, [,], \langle, \rangle, \text{`,'}, (,)\}, \\ & Q, \\ & S) \end{aligned}$$

and Q consists of the productions

$$\begin{aligned} S &\rightarrow R \mid \text{`}/_l\ XS\text{'} \mid \text{`}*_m\ XS\text{'} \\ X &\rightarrow XS \mid S \\ R &\rightarrow \{A\} \\ A &\rightarrow B \mid [+_n\ YA] \mid [-_o\ YA] \mid [\%_p\ YA] \mid [@_q\ YA] \\ Y &\rightarrow YA \mid A \\ B &\rightarrow \langle P\mathbf{m}_j\rangle \mid \langle \mathbf{a}_k P\mathbf{m}_j\rangle \mid \langle P\rangle \mid \langle \mathbf{a}_k P\rangle \\ P &\rightarrow \mathbf{p}_i \mid (\#_r\ ZP) \mid (=_s\ ZP) \\ Z &\rightarrow ZP \mid P \end{aligned}$$

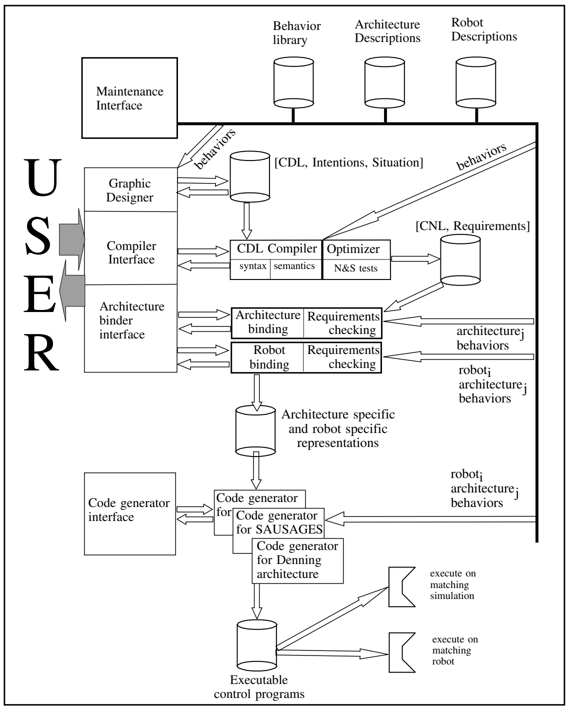

Figure 1: Architecture diagram of configuration development system

Where

- S is a society
- X is a list of one or more societies
- R is a single robot
- A is a behavioral assemblage
- Y is a list of one or more assemblages
- B is a primitive sensorimotor behavior
- P is a perceptual module or a coordinated perceptual group
- Z is a list of one or more perceptual modules

- $\mathbf{p}_i, i \in$ *natural numbers* is an instance of a perceptual process
- $\mathbf{m}_j, j \in$ *natural numbers* is an instance of a motor process
- $\mathbf{a}_k, k \in$ *natural numbers* is an instance of an active perception motor process
- $/_l, l \in$ *natural numbers* is an instance of a caste (heterogeneous) society operator
- $*_m, m \in$ *natural numbers* is an instance of a team (homogeneous) society operator
- $+_n, n \in$ *natural numbers* is an instance of an assemblage cooperation operator
- $-_o, o \in$ *natural numbers* is an instance of an assemblage competitive operator
- $\%_p, p \in$ *natural numbers* is an instance of an assemblage sequencing operator
- $@_q, q \in$ *natural numbers* is an instance of the generic assemblage coordination operator
- $\#_r, r \in$ *natural numbers* is an instance of a perceptual fusion operator
- $=_s, s \in$ *natural numbers* is an instance of a perceptual sequencing operator
- ‘ ’ delineates societies
- { } delineates agents (robots)

- [ ] delineates coordinated assemblages
- ⟨ ⟩ delineates primitive sensorimotor behaviors
- ( ) delineates a group of coordinated perceptual modules.

Sensors are explicitly represented to allow parameterization and to facilitate hardware binding. Perceptual modules function as virtual sensors which extract features from one or more sensation streams and generate as output a stream of features (individual percepts). Motor modules use one or more feature streams (perceptual inputs) to generate an action stream (a sequence of actions for the robot to perform). Perceptual coordination is the process of linking one or more perceptual modules to motor modules and is partitioned into three categories[3]: sensor fission, action-oriented perceptual fusion, and sensor fashion. Active perception utilizes a special motor module which generates an action stream to modify the information the sensor is providing. A primitive behavior consists of one or more perceptual modules and a motor module generating a stream of actions based on perceptual inputs. An assemblage can be treated as a single sensorimotor behavior even though it may be recursively composed of many primitive behaviors and coordination strategies. Each individual robot is controlled by a single assemblage. Societies of robots come in three types; trivial, homogeneous teams, and heterogeneous castes.

3.2 The Configuration Network Language (CNL)

The CDL compiler generates a Configuration Network Language (CNL) specification of the configuration as its output. CNL is a hybrid dataflow language[14] using large grain parallelism where the atomic units are arbitrary C++ functions. CNL adds dataflow extensions to C++ which eliminate the need for users to include communication code. A compiled extension to C++ was chosen to allow verification and meaningful error messages to assist casual C++ programmers in constructing behaviors. The separation of the code generator from the CDL compiler permits incremental development and testing of the design tools as well as simplifying retargeting.

A CNL configuration can be viewed as a directed graph, where nodes are threads of execution and edges indicate dataflow connections between producer nodes and consumer nodes. Each node in the configuration is an instantiation of a C++ function, forked as a lightweight thread using the C-Threads package[20] developed at Georgia Tech. UNIX processes are examples of heavyweight threads which use the operating system for scheduling. Lightweight threads are generally non-preemptive and scheduled by code linked into the user's program. All lightweight threads execute in the same address space and can share global variables. The advantage of lightweight threads is that a task switch takes place *much* faster than between heavyweight threads, allowing large scale parallelism. Current robot configurations are using around 50 threads with little overhead, while that many UNIX processes is not feasible. Code for thread control and communication synchronization is explicitly generated by the CNL compiler and need not be specified by the user.

The use of communicating processing elements is similar to the Robot Schemas (RS)[16] architecture which is based on the port automata model. The major differences are that RS uses synchronous communication while CNL is asynchronous to support multiprocessing; and RS is an abstract language while a CNL compiler has been developed. Both use the notion of functions using data arriving at input ports to compute an output value, which is then available for use as inputs in other functions.

3.2.1 The Behavior Development Process

To support development of abstract configurations, CDL descriptions of each new primitive behavior must be created. Grounding these abstract behaviors in executable code requires development of a corresponding CNL behavior. These behaviors form a component library used during construction of skill assemblages.

Figure 2 shows an example schema-based CNL procedure implementing the **move_to_goal** behavior. The behavior receives the relative location of the goal and a constant defining how close the vehicle should attempt to get to the goal. The behavior uses this information to compute a desired movement for the vehicle.

Figure 3 shows the C++ code generated from this description. Notice that all thread scheduling and data communications code is automatically generated by the CNL compiler, simplifying the developers task. The right-hand box shows the task control block generated by the compiler to maintain input/output connection information. Keep in mind that there may be several instances of this procedure executing, each with its own

```
procedure Vector MOVE_TO_GOAL with
   Vector  goal_relative_loc;
   double  success_radius;
header
   // optional user initialization code
body
   // user C++ code
   if( len_2d(goal_relative_loc) > success_radius )
   {  // generate a vector towards the goal
      output = goal_relative_loc;
      unit_2d(output);
   }
   else
   {  // return a zero vector if within the success circle
      VECTOR_ZERO(output);
   }
pend
```

Figure 2: Example **move_to_goal** behavior in CNL

instance of this task control block. Code is generated by the CNL compiler to wait for the input parameters to be updated and then copy the values to the local address space. The availability of fresh input data is checked using sequence numbers. The thread will block until fresh data arrives on all input ports receiving data from other nodes. The content of the data depends on the semantics of the node. In this case, **goal_rel_loc** will be an egocentric vector pointing towards the goal generated by another node (e.g., **detect_goal_with_shaft_encoders**), and **success_radius** is a constant denoting how close the behavior should try to get to the goal. The user's code is executed to compute a new output (named *output*). Code emitted by the compiler posts the new output and blocks until new inputs are received.

3.3 Skill Assemblage Specification

Skill assemblages encapsulate a particular skill or ability where a skill is a higher level construction than a behavior[17]. A skill such as "follow road" will include several behaviors (e.g., **stay_on_road**, **move_ahead**, and **avoid_static_obstacles**), other skill assemblages (e.g., **avoid_dynamic_obstacles** and **maintain_formation**), coupled with suitable coordination mechanisms permit the multi-robot group to function as an integrated unit. The low-level grounded behaviors described earlier provide an atomic component for reuse.

Construction of the skill assemblages by the system designer occurs using a graphical editor developed as part of this research. Figure 4 shows a screen snapshot of a simple *wander* assemblage loaded in the editor. The output of the **detect_obstacles** perceptual schema is fed into two motor behaviors: **probe**, which encourages the robot to move towards free space areas in the direction its already heading; and **avoid-static-obstacle**, which prevents collisions. **Probe** is supplemented with perceptual data from a heading perceptual algorithm. The **noise** behavior generates a random direction periodically to ensure coverage of a broad area. The outputs of the three active motor schemas (**probe**, **noise**, and **avoid-static-obstacle**, are combined using the **sum** coordination operator and form the output of the assemblage. Figure 5 shows an instance of the wander configuration parameterized for use on our Denning robots. Obstacle sensor information is provided from the ring of ultrasonic sensors on the robot, while shaft encoders maintain vehicle heading. Constant parameters for each behavior and coordination mechanism are shown within the objects themselves.

The overall design process for an assemblage is as follows:

- Components are selected from a library menu and placed within the workspace.
- Dataflow connections are added by clicking on the corresponding input/output arrows.

```
void MOVE_TO_GOAL(int parm)
{
   Vector  output;
   struct T_MOVE_TO_GOAL *parms = (struct T_MOVE_TO_GOAL *)parm;
   double success_radius;
   Vector goal_rel_loc;
   /******** start of user header ********/
   /******** end of user header ********/
   while(1)
   {
      if( parms->success_radius_chk)
      {
         while( parms->success_radius_last_seq ==
        *parms->success_radius_seq)
            cthread_yield();
         parms->success_radius_last_seq = *parms->success_radius_seq;
      }
      success_radius = *parms->success_radius;
      if( parms->goal_rel_loc_chk)
      {
         while(parms->goal_rel_loc_last_seq == *parms->goal_rel_loc_seq)
            cthread_yield();
         parms->goal_rel_loc_last_seq = *parms->goal_rel_loc_seq;
      }
      goal_rel_loc = *parms->goal_rel_loc;
      /******** start of user body ********/

      if( len_2d(goal_relative_loc) > success_radius )
      { // generate a vector towards the goal
        output = goal_relative_loc;
        unit_2d(output);
      } else   // return a zero vector if within the success circle
        VECTOR_ZERO(output);

     /******** end of user code ********/
     parms->output = output;
     parms->seq++;
     mutex_lock(_out_count_lock);
      condition_wait(_new_cycle,_out_count_lock);
      mutex_unlock(_out_count_lock);
   }
}
```

```
struct T_MOVE_TO_GOAL
{
   double *success_radius;
   int success_radius_chk;
   int *success_radius_seq;
   int success_radius_last_seq;

   Vector *goal_rel_loc;
   int goal_rel_loc_chk;
   int *goal_rel_loc_seq;
   int goal_rel_loc_last_seq;

   int seq;
   Vector  output;
}
```

Figure 3: Code generated by CNL compiler for move_to_goal (reformatted for readability)

- Specific behavioral parameters are then entered using the mouse and keyboard.

The completed skill assemblages can be saved as additions to the component library for reuse in subsequent missions.

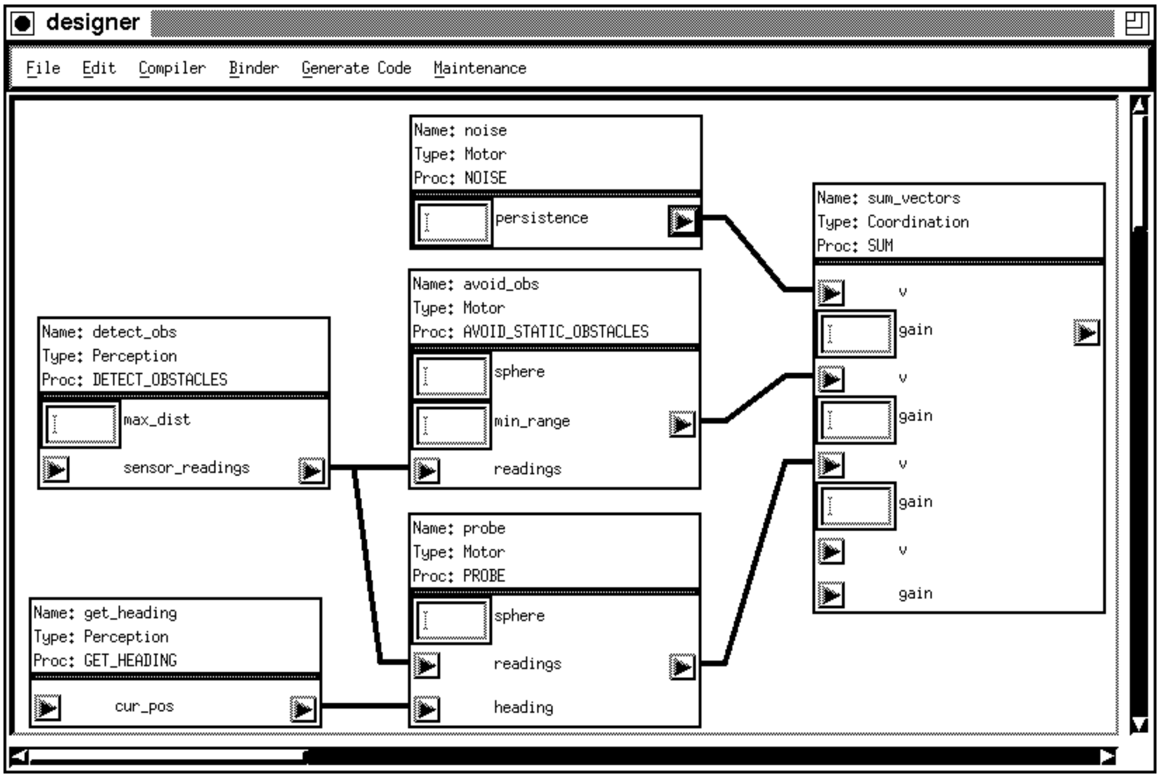

Figure 4: Abstract wander skill assemblage

3.4 Specifying Temporal Chains of Assemblages

Skill assemblages consisting of temporal chains of other skill assemblages are constructed using sequenced coordination operators. Currently, the operators must be specified manually, although development of a graphic editor is planned. The assemblage is constructed from a selected group of skill assemblages with the appropriate sequenced coordination operator and perceptual triggers.

Figure 6 shows the forage assemblage[4] loaded in the graphic editor. The Wander assemblage is the component representation of the assemblage shown in Figure 5. Notice that the inputs and output of the wander component match the unconnected inputs and output of the assemblage. Detailed views of the **Acquire** and **Deliver** assemblages are not shown, but are similar in complexity. The two perceptual triggers **pt_attractor_present** and **pt_holding_attractor** control transitions between the three operating states. Figure 8 shows the state diagram for the forage coordination operator. Figure 7 shows an instance of the forage assemblage parameterized for use on our Denning robots.

3.5 Configuration Specification

The configuration is the top-level robot assemblage which has been parameterized with the actuator modules. The configuration is constructed from a selected group of skill assemblages using techniques similar to those used to construct the assemblages themselves.

Figure 9 shows the complete forage configuration loaded into the graphic editor, parameterized for execution

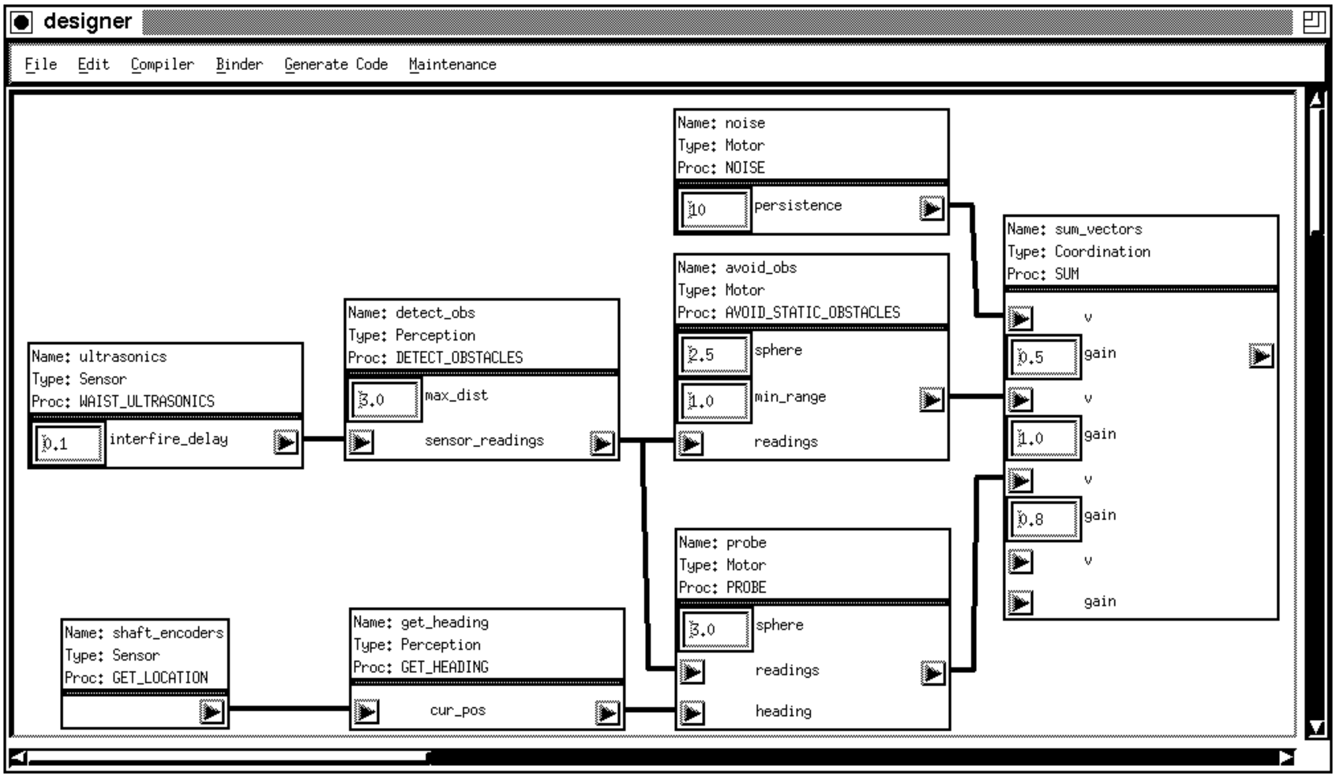

Figure 5: Parameterized instance of wander skill assemblage

on Denning robots. The Forage assemblage is the component representation of the assemblage shown in Figure 7. The output of the assemblage will be sent to the robot for execution.

3.6 Specifying Multiple Robot Societies

Currently, coordination of societies of multiple robots occurs via the mission coordination operator and the Mission Description Language (Section 4.2). Although this centralized society coordination has proven useful within the military missions that are being developed for the ARPA UGV program, effort is underway to distribute the society coordination among the robots while retaining the utility of the operator console. It remains difficult to provide centralized command and control over distributed systems.

Within the Mission Description Language (MDL), societies of robots are called **units**. A unit is a recursive structure where units can be composed of other units. The following example constructs two units with two robots each (**team-1** and **team-2**) and a four robot unit (**group**).

```
UNIT <group> (<team-1> ROBOT ROBOT) (<team-2> ROBOT ROBOT)
```

The three names can then be used to target commands to specific groups of robots. Figure 10 shows a configuration for a unit of forage robots with three members. The coordination operator is responsible for coordinating the activities of the group as a whole, including any needed synchronization.

4 Executing Missions

To execute missions by simulation or with real robots, we have developed *MissionLab*. Part of *MissionLab* is an operator console program which displays the simulation environment and the locations of all robots (simulated or

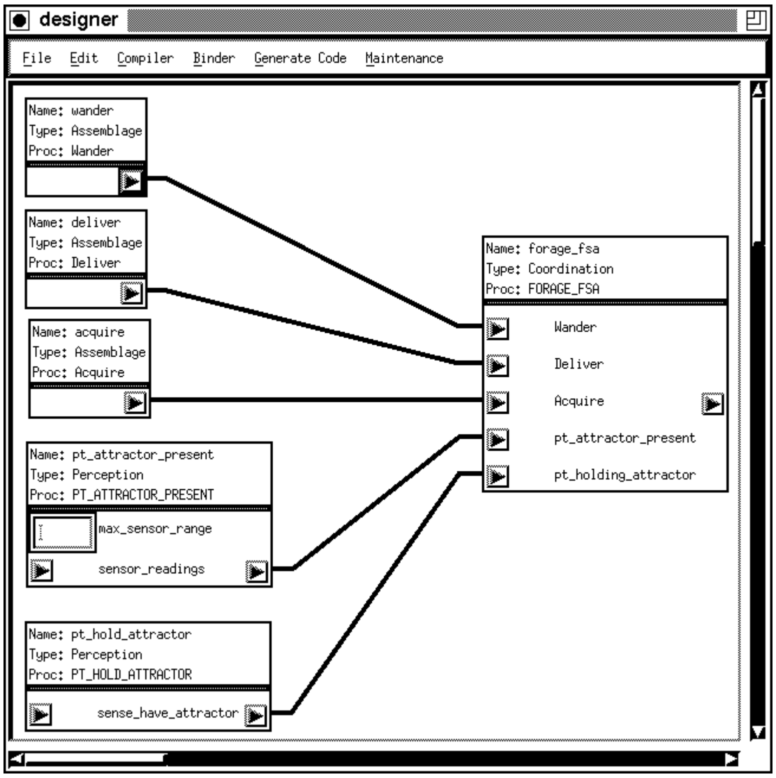

Figure 6: Abstract forage assemblage

real). *MissionLab* also includes other "robot programs" which simulate robots or control real robots. *MissionLab* uses "mission description" files as scripts for missions.

4.1 *MissionLab*

Figure 11 shows the *MissionLab* executing a simulation. The large area with various things drawn in it is the main display area. Within the display area robots, obstacles, and other features are visible. The solid round black circles are obstacles. The four robots are moving across the middle of the display area in roughly a diamond formation. More details about the type of mission displayed in the figure are explained in the next section. The command interface in the lower right part of Figure 11 allows the operator to control the execution of the mission. The steps of the mission are displayed as they execute. For more detail on the operation of *MissionLab*, see [6].

4.2 Mission Description Language

Complex robot missions require construction of temporal chains of behaviors (e.g., moving to door, moving through door, closing door, lock door, etc.). These temporal chains can be constructed by adding preconditions to behaviors such that finishing one behavior makes changes to the world that trigger the next behavior in the chain. This is rather unwieldy and prone to inadvertent loops[18]. The technique of temporal sequencing[2] makes the temporal sequencing explicit in the form of a Finite State Automaton (FSA)[11]. This provides sufficient expressive power for most missions. However, specifying a FSA still requires programming knowledge from the user. To increase the expressive power of the sequenced operator while reducing the requirements on the user, the mission coordination operator has been developed.

The mission coordination operator functions like the FSA-based sequenced coordination operator but instead of specifying an FSA, the user specifies the temporal chain (the mission) using a domain-specific language with high-level primitives and mnemonic names. It is important to note that the robot still is running reactively. The

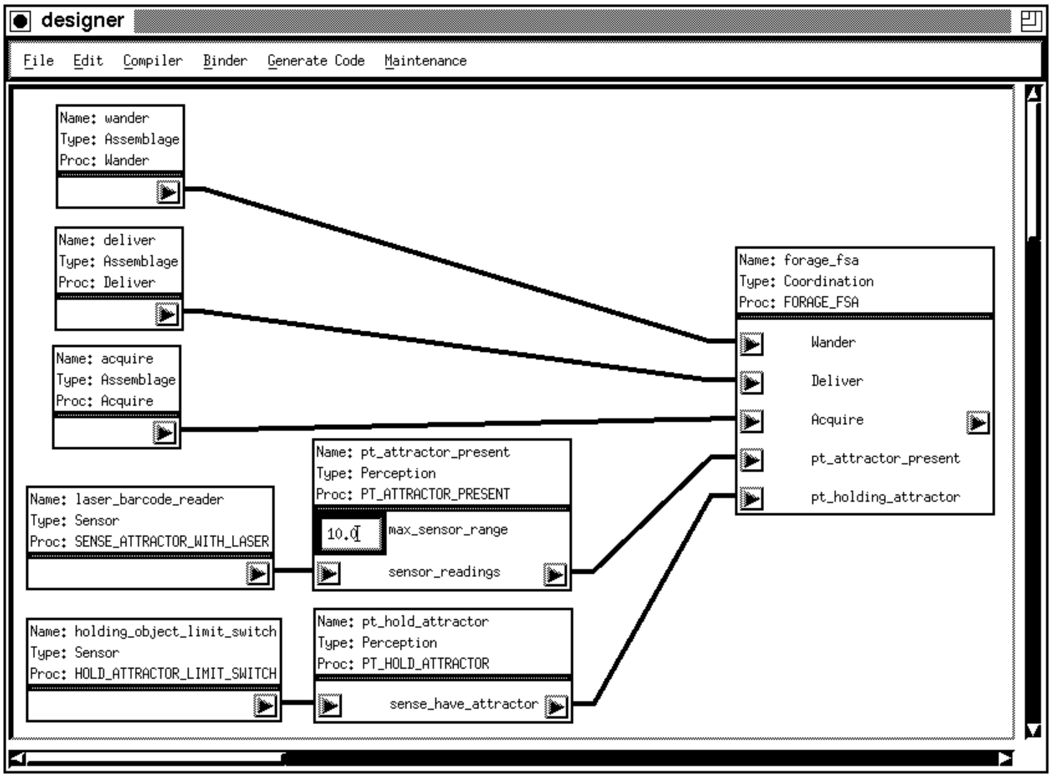

Figure 7: Parameterized instance of forage assemblage including perceptual triggers

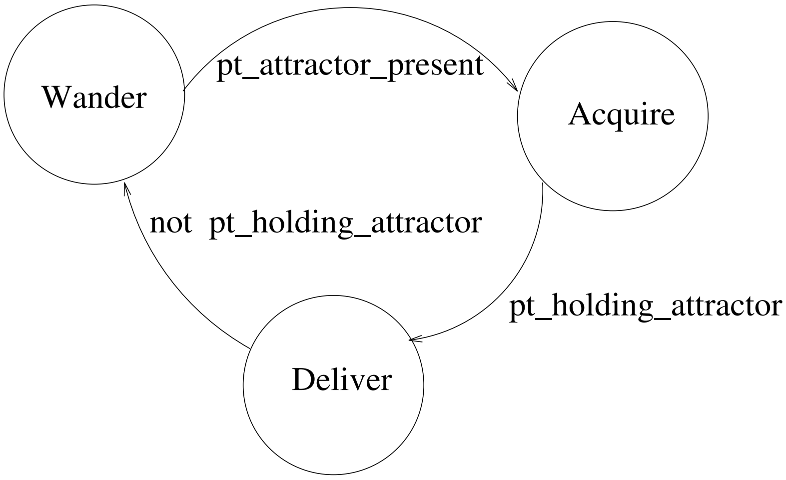

Figure 8: State Diagram for Forage Assemblage

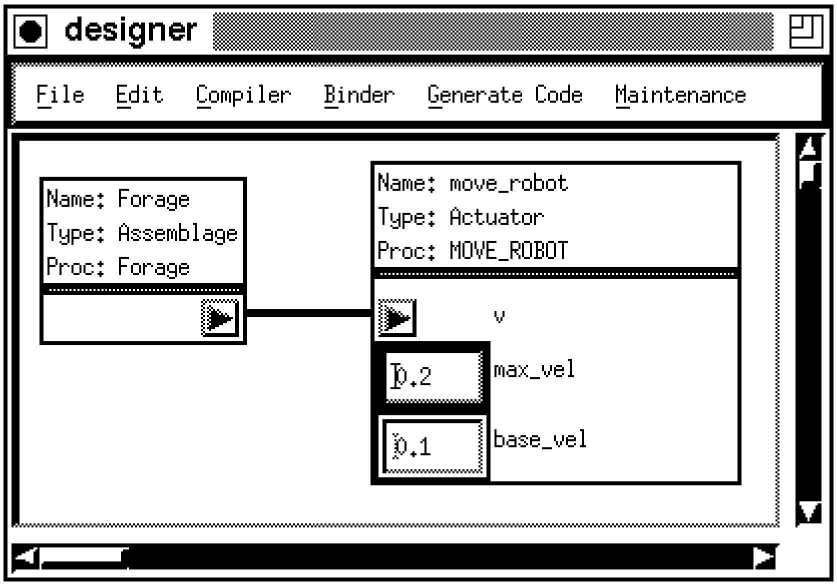

Figure 9: Robot forage configuration loaded in the graphic editor

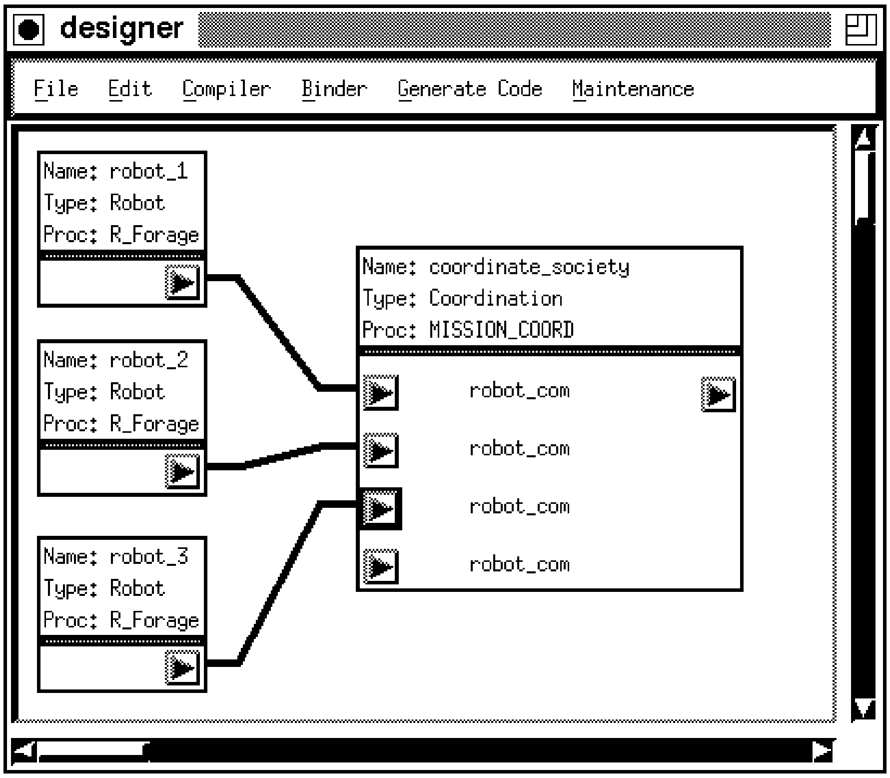

Figure 10: Configuration for a robot society with three members

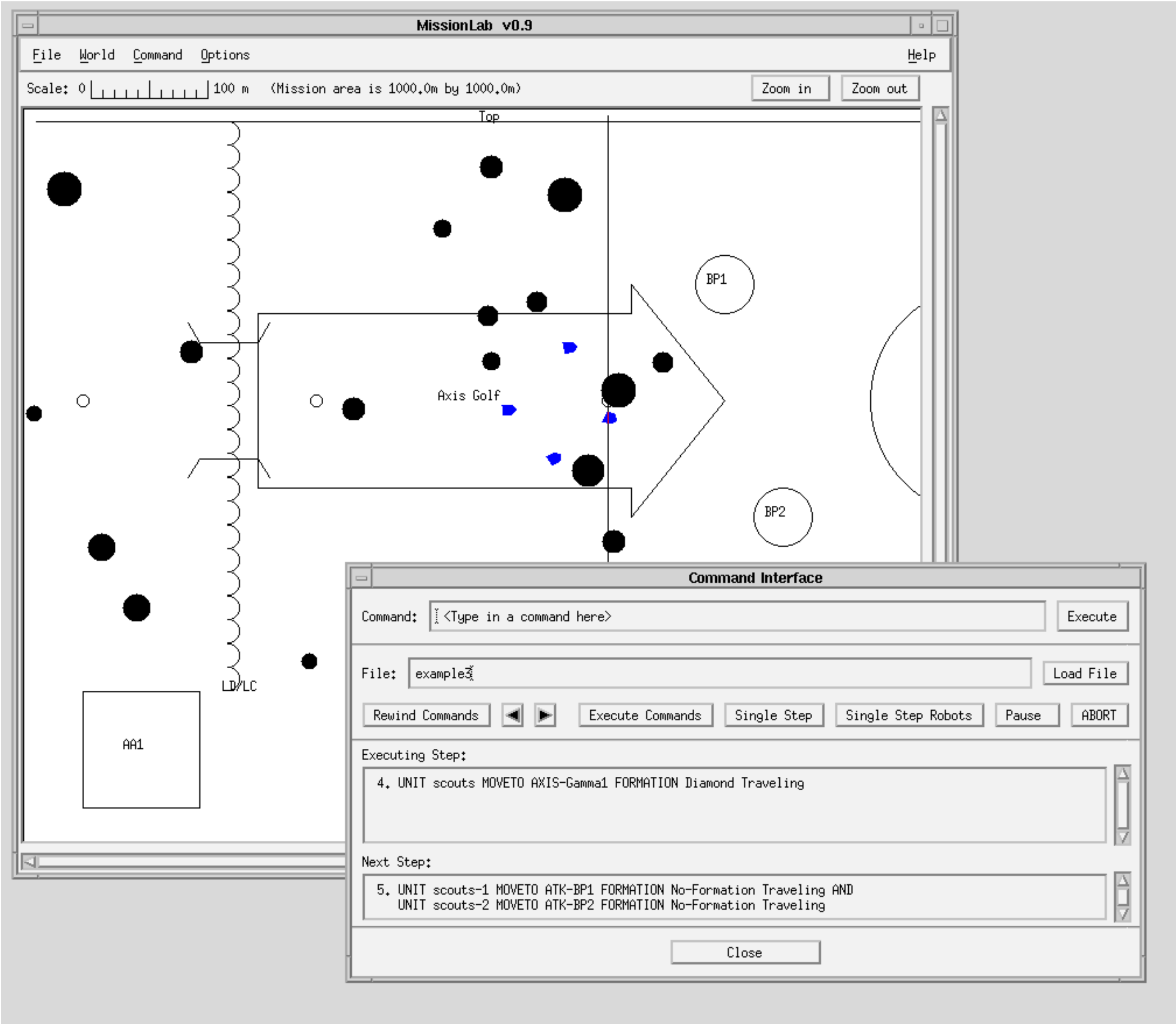

Figure 11: Example scenario being executed in *MissionLab*

mission coordination operator notifies the robot to activated the appropriate skill assemblage. It then waits until the robot indicates it has completed the task.

The mission coordination operator communicates with the operator console to allow the mission to be entered interactively or predefined missions to be executed from saved files. Currently, the mission language interpreter is resident on the operator console and functions as the societal coordination operator. In future versions, the interpreter will be moved onto the robots to better mesh with our schema-based control paradigm[1]. This move will necessitate communication facilities which will allow the operator console to monitor and perhaps modify execution of the mission scenarios, as well as communication facilities that will allow the robots to coordinate directly with each other.

The mission scenario language and interpreter permit the specification of complex multiagent missions in a structured relatively user-friendly language. An example set of commands is shown in Figure 12.

```
MISSION NAME "Demo C simulation"
SCENARIO "Demo-C"

OVERLAY democ.odl

UNIT <scouts> (<scouts-1> ROBOT ROBOT) (<scouts-2> ROBOT ROBOT)

COMMAND LIST:
  0. UNIT scouts START AA-AA1 0 20
  1. UNIT scouts OCCUPY AA-AA1 FORMATION Column
  2. UNIT scouts MOVETO ATK-AP1 FORMATION Column
  3. UNIT scouts OCCUPY ATK-AP1 FORMATION Diamond
  4. UNIT scouts MOVETO PP-Charlie FORMATION Column
  5. UNIT scouts MOVETO PP-Delta1 FORMATION Column
  6. UNIT scouts MOVETO AXIS-Gamma1 FORMATION Diamond
  7. UNIT scouts-1 MOVETO ATK-BP1 AND
     UNIT scouts-2 MOVETO ATK-BP2
  8. UNIT scouts-1 OCCUPY ATK-BP1 AND
     UNIT scouts-2 OCCUPY ATK-BP2
  9. UNIT scouts MOVETO OBJ-Tango FORMATION Wedge
 10. UNIT scouts OCCUPY OBJ-Tango FORMATION Diamond
 11. UNIT scouts STOP
```

Figure 12: Example Mission Scenario Commands

There are several features to note regarding this mission description format. The preamble (before the line containing `COMMAND LIST:`) sets up the environment for the mission scenario. In the preamble, environmental details are specified such as the name and location of the place where the simulation or actual run are to take place. The `OVERLAY` command instructs the console to load a file of overlay data which includes all the military control features necessary to accomplish the mission. In military usage, an overlay is typically a transparent sheet with markings which is laid on top of a map to indicate the positions and extents of objects or positions necessary to execute the mission. Control measures include objects such as roads, assembly locations, boundaries, and objectives. For more details about the simple overlay description language we created for this purpose, refer to [6]. The resulting map is shown in Figure 13.

The UNIT command defines the unit `scouts`. This unit is composed of two subunits, `scouts-1` and `scouts-2`, each of which is composed of two generic robotic vehicles called `ROBOT`. The list of commands (below `COMMAND-LIST:`) is a series of steps to be done as part of the mission. The preliminary step, Step 0:

```
0. UNIT scouts START AA-AA1 0 20
```

starts the robots in unit `scouts` and displays them on the screen at the specified starting position (Assembly Area "AA1"). This is shown in Figure 14. Notice the four robots which are shaped like solid rectangles with one

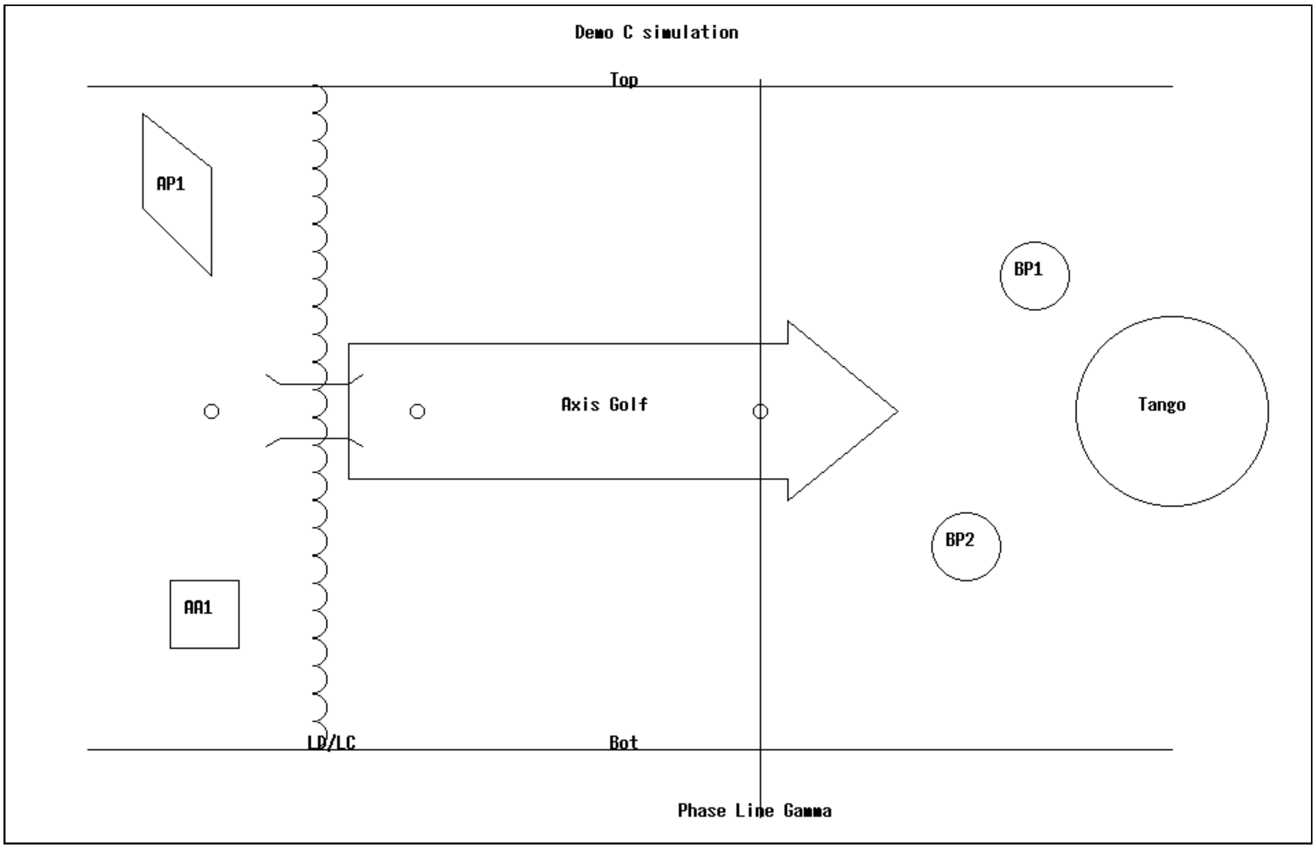

Figure 13: Example mission overlay

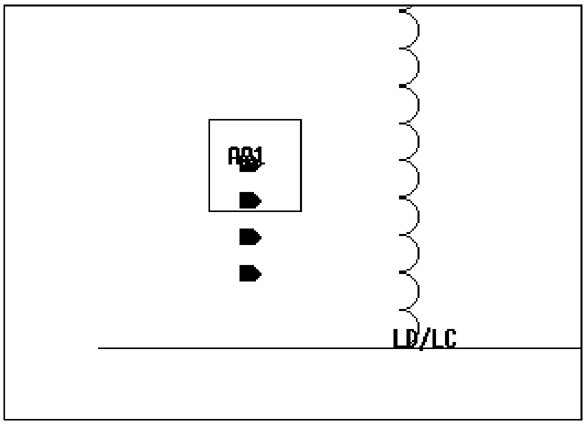

Figure 14: Mission after starting

pointed end on the right. The pointed end indicates the direction the vehicle is pointing.

"Starting" the unit involves executing a robot program in its own process for each robot in the unit. Each robot program is instructed where to position itself (in the simulation environment) using command-line arguments. At this point, each robot program has no active assemblages.

After the "Starting" command, the main commands to accomplish the mission begin. Step 1:

```
1. UNIT scouts OCCUPY AA-AA1 FORMATION Column UNTIL TIMEOUT 10
```

instructs the unit to occupy the starting location in column formation until it receives operator approval to continue. When this command is executed, the operator console constructs a data structure with the information about what assemblage to activate (and necessary parameter data) and sends it in a message to the robot programs for each of the robots in the unit `scouts`. Each robot program examines the message and decides what it should do to satisfy the command. In this case, an assemblage is activated which knows how to "Occupy" and includes behaviors to maintain formations. Once the formation has been achieved in the correct location, the robot programs send messages to the operator console that they have completed the command. At this point, the operator console pops up a "Proceed?" dialog box, allowing the operator to give or deny permission for the unit to proceed. This is shown in Figure 15 (A timeout can also be specified for continuing automatically after a set

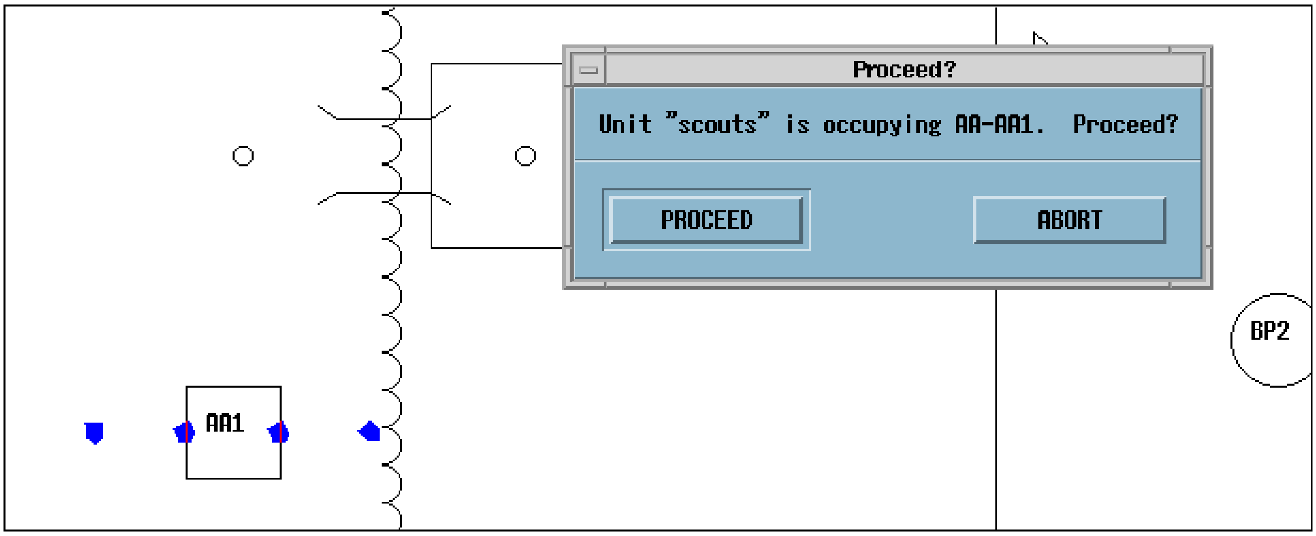

Figure 15: Mission after starting

amount of time or at a specific time.) The purpose of interacting with the operator at this point is to simulate the operation of military missions where units often check with the commander before continuing some stage of a mission.

Step 2 instructs the robots in unit `scouts` to move to attack position "AP1" in column formation:

```
2. UNIT scouts MOVETO ATK-AP1 FORMATION Column
```

As in the previous step, and appropriate assemblage is activated which knows how to `MOVETO` the specified location using column formation. This step is shown in mid-execution in Figure 16. Steps 3 through 6 move the robot through a series of way-points in various formations.

Notice that in Step 7,

```
7. UNIT scouts-1 MOVETO ATK-BP1 AND
   UNIT scouts-2 MOVETO ATK-BP2
```

the unit `scouts` is subdivided into two subunits `scouts-1` and `scouts-2` and each subunit has its own separate command separated by an `AND`. The resulting split is shown in Figure 17. In this case, both subunits have `MOVETO`

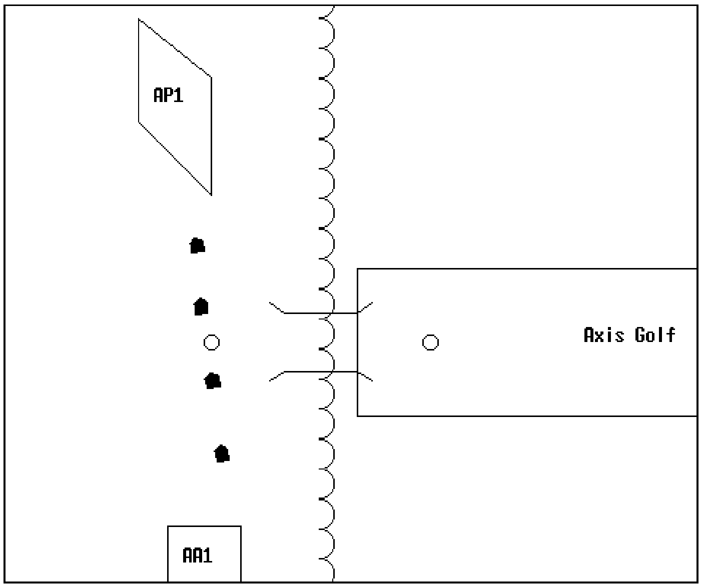

Figure 16: Unit moving to ATK-AP1

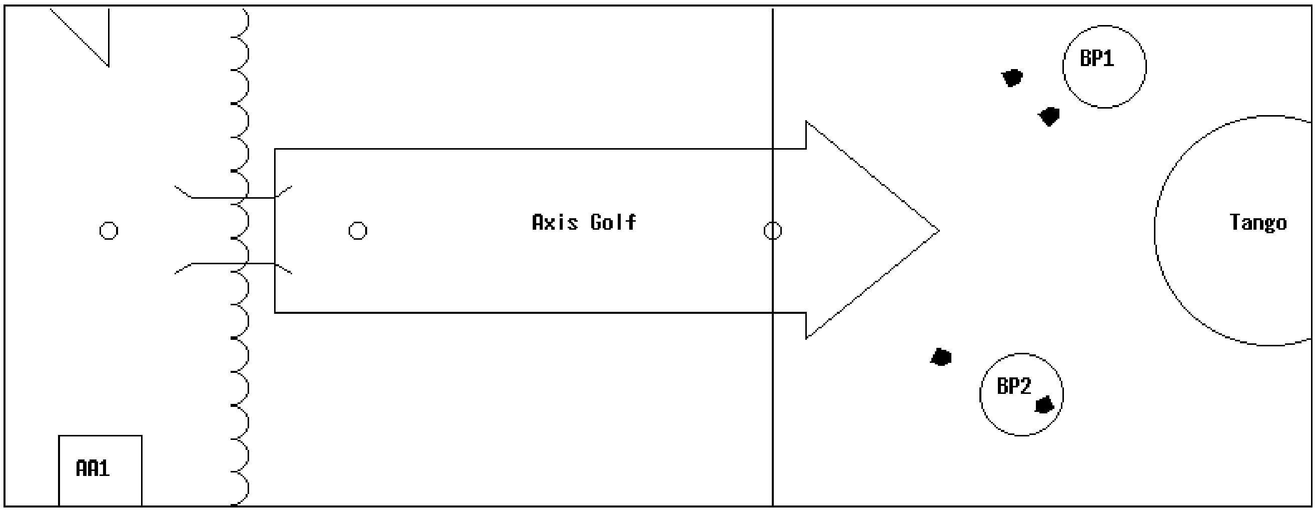

Figure 17: Unit split into subunits

commands, although any command could have been given. The two robots in subunit `scouts-1` are moving towards BP1 and the two robots in `scouts-2` are moving towards BP2. These two commands are executed in parallel by the robot programs for each subunit. Both subunits must finish their commands before the step is complete. The ability to deal with units as a whole or in various subgroups is an important feature of the multiagent nature of the system.

Once the objective has been achieved (in Step 10), the mission is terminated with Step 11:

```
11. UNIT scouts STOP
```

which instructs the executing robot programs in unit `scouts` that the mission is complete. The process for that robot program is then terminated. Further details about the command description file format can be found in [6].

5 Results with Denning Robots

5.1 One robot

Figure 18 shows *MissionLab* with an overlay representing the Georgia Tech Mobile Robot Lab. The robot is shown in its starting location, in the lower left. The overlay file specifying this environment is shown in Figure 19. The SCENARIO command names the environment. The SITE command provides a description of the environment. The Boundary marks the walls of the Mobile Robot Lab (units in Meters). The Gap specifies the door to the lab. Two passage points (PP) (shown as circles in Figure 18) were chosen arbitrarily to use as targets for MOVETO commands in the mission.

Figure 20 shows the mission description file used in these experiments. It commands the robot to move to the far right circle back to the middle circle, and then return to the starting location. The keyword/value pairs specify configuration parameters.

Figure 21 shows a screen snapshot of the mission executing in simulation. The trail left by the robot shows it successfully completed the mission. Figure 22 shows a screen snapshot of the same mission executing on the Denning. This was create by changing the **robot_type= "SIMULATION"** value to **robot_type= "MRV2"**. The robot control executable then attaches to the actual Denning robot instead of the simulation server. The filled circles represent obstacles detected by the Denning robot. The differences in trajectories between the actual run and the robot run are largely due to the detection of un-modeled obstacles that repulse the robot (the black circles in Figure 22). Figure 23 shows pictures of the robot while completing the mission.

6 Conclusions and future work

This paper describes on-going research in how to specify and control societies of robots while they perform multiagent tasks. Several useful features of our research are described. The ability to recursively construct a high-level behavior (or assemblage of behaviors) based on more primitive behaviors is a powerful tool. It will allow simple creation and reuse of more generalized and useful behaviors. The CDL and CNL compilers permit the construction of useful high-level behavior assemblages using a graphical interface and associated compilers that reduces the amount of hands-on programming necessary.

Using the *MissionLab* simulation system, an operator can run a variety of missions which activate real or simulated robots, instruct them in how to execute the mission step by step, and display the positions of the robots and the environment in which they move. The mission description language used to specify missions executed by *MissionLab* is designed to deal with multiagent teams cooperating in multiagent tasks. To validate the usefulness of the concepts and implementations presented in this paper, simulated and real runs of the same mission are presented.

Next year we plan to demonstrate this technology on ARPA UGV Demo C. This demonstration involves operating a pair of automated off-road vehicles to accomplish several scouting tasks. This team of vehicles will

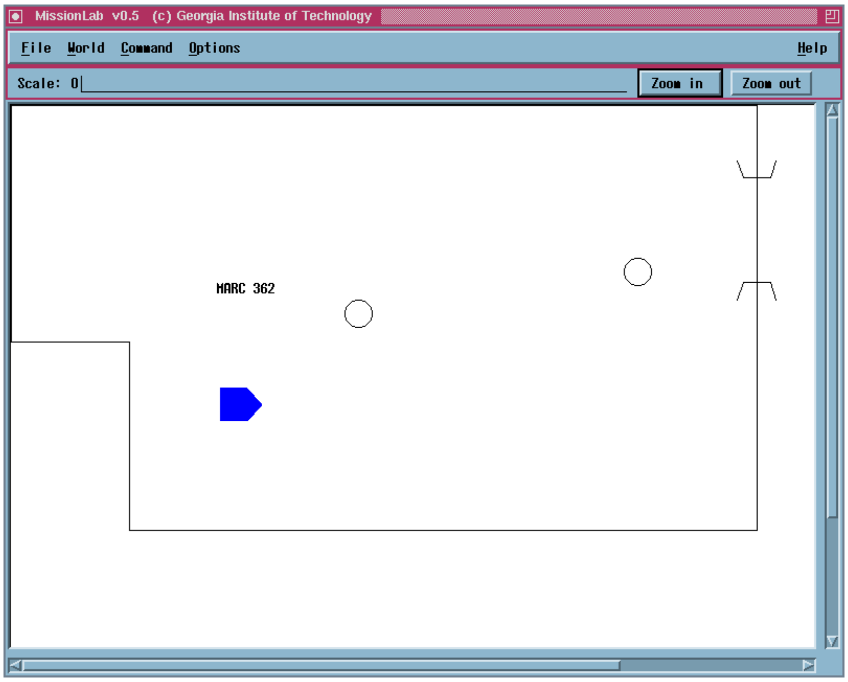

Figure 18: *MissionLab* with the overlay loaded

```
SCENARIO "example1"
SITE "Robot Lab, MARC 362"
CONTROL MEASURES:

Boundary "MARC 362" 0 0  10.7 0  10.7 6.1  1.7 6.1  1.7 3.4  0 3.4  0 0
Gap Door 10.5 1.8    10.9 1.8  1.5
PP DoorWay 9.0 2.4 0.4
PP Middle 5.0 3.0 0.4
```

Figure 19: Mobile Robot Lab Overlay

```
MISSION NAME "Single Denning MRV2 Robot Simulation"
OVERLAY "robot_lab.odl"
SP StartPlace 3.4 4.26   --   Note: x means EAST, y means NORTH

NEW-ROBOT stimpy-the-robot "robot"     (robot_type= "SIMULATION",
tty_num= 0,
"timeout"= 10,
echo_tty = 0,
move_to_goal_success_radius = 0.3,
navigation_success_radius = 0.3,
avoid_obstacle_sphere = 1.25,
navigation_move_to_goal_gain = 1.0,
navigation_avoid_obstacle_gain = 0.7,
lurch_mode = 1,
base_velocity = 0.1,
max_velocity = 0.1
)

UNIT <unit-stimpy> stimpy-the-robot

COMMAND LIST:

 0. UNIT unit-stimpy START StartPlace 0 20
 1. UNIT unit-stimpy MOVETO DoorWay
 2. UNIT unit-stimpy MOVETO Middle
 3. UNIT unit-stimpy MOVETO StartPlace
 4. UNIT unit-stimpy OCCUPY StartPlace
 5. UNIT unit-stimpy STOP
```

Figure 20: Script file used in missions

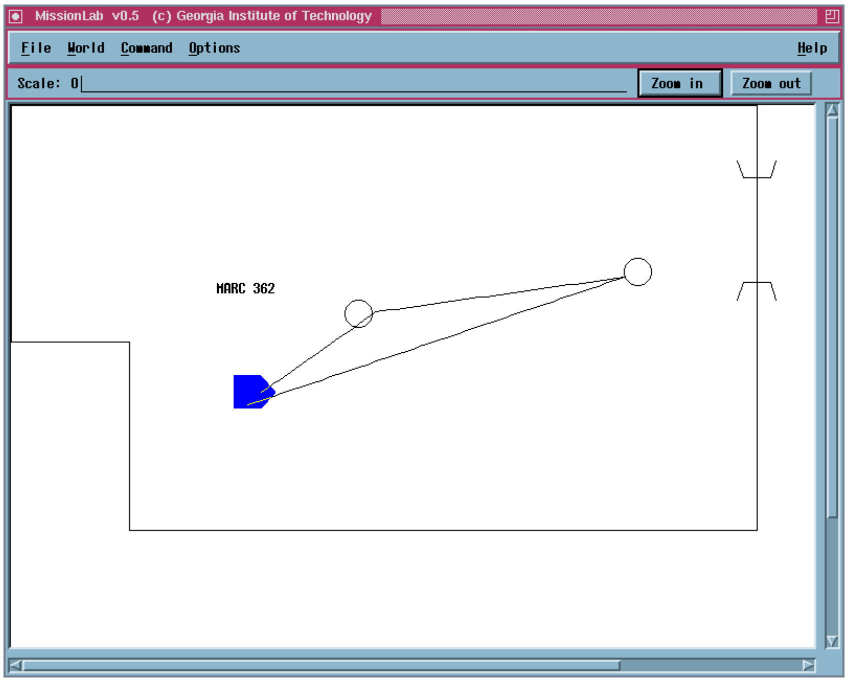

Figure 21: Screen snapshot of the mission executing in simulation

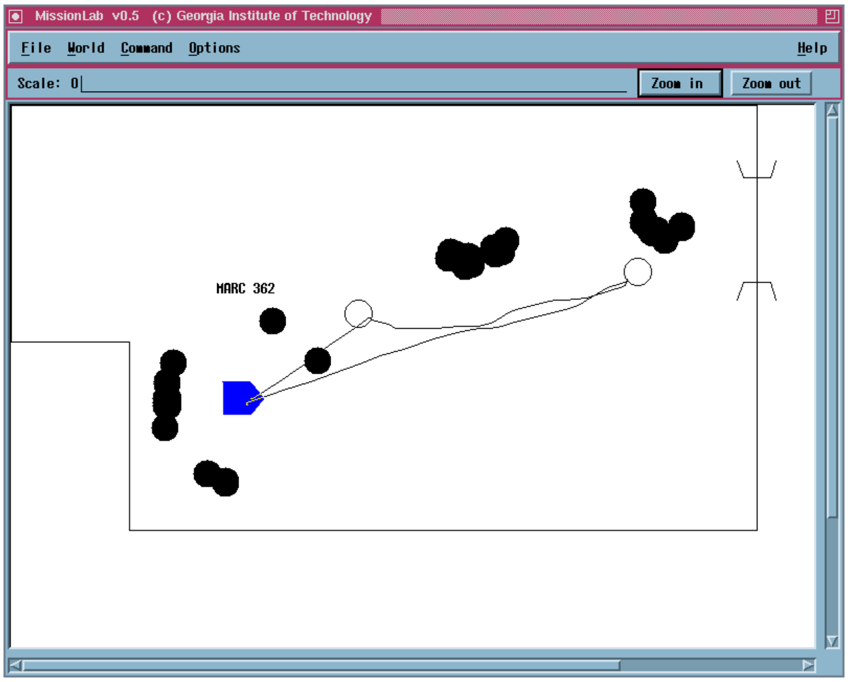

Figure 22: Screen snapshot of the mission executing on the Denning

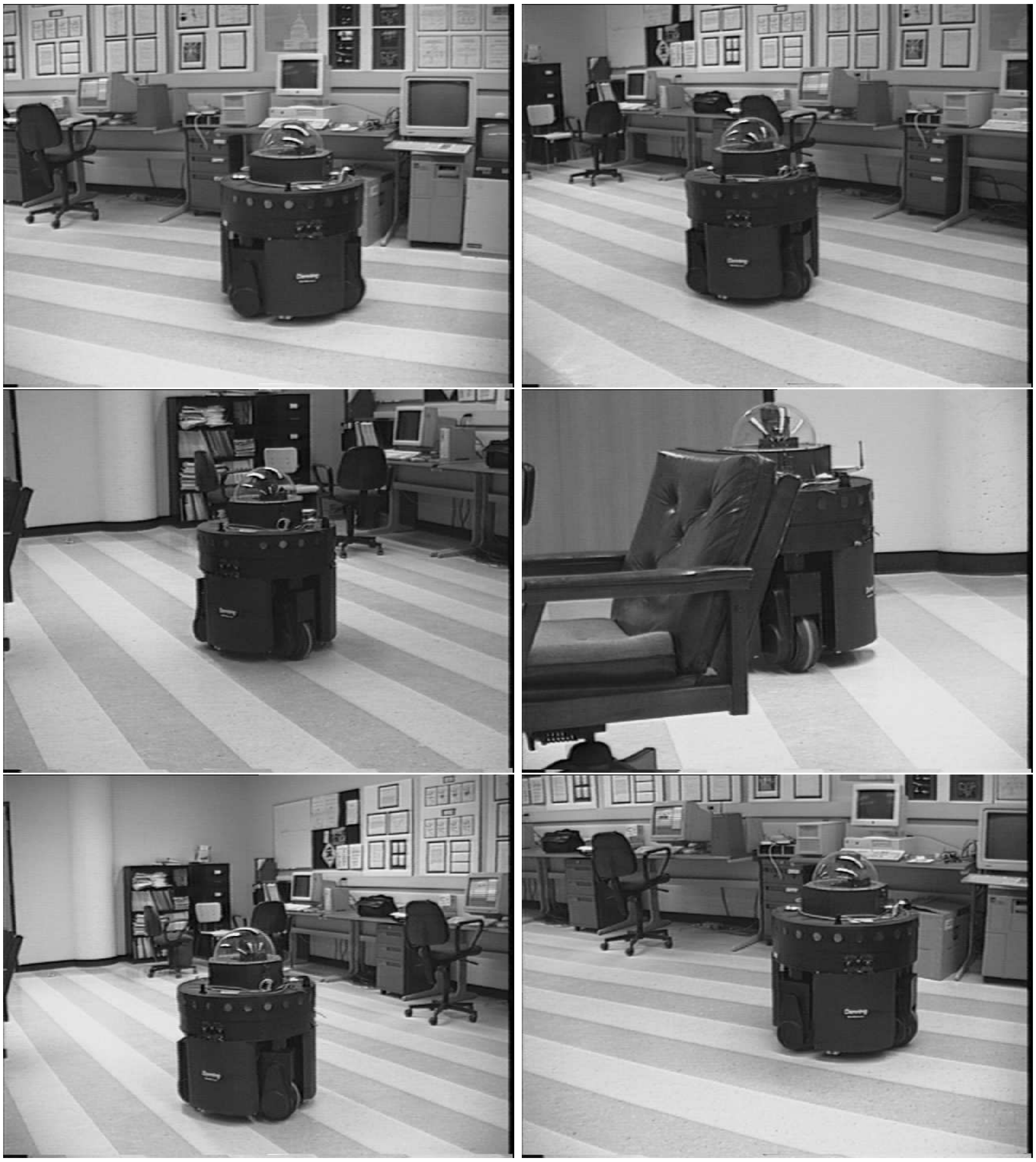

Figure 23: Pictures of the robot executing the mission

also be controlled using software developed using different approaches. It will be a useful exercise to compare the utility and operation of the two different approaches.

Acknowledgments

This research is funded under ONR/ARPA Grant # N0001494-1-0215.

References

[1] Arkin, R.C., "Motor Schema-Based Mobile Robot Navigation", *International Journal of Robotics Research*, Vol. 8, No. 4, August 1989, pp. 92-112.

[2] Arkin, R.C. and MacKenzie, D., "Temporal Coordination of Perceptual Algorithms for Mobile Robot Navigation", *IEEE Transactions on Robotics and Automation*, Vol 10, No. 3, June 1994, Pg 276-286.

[3] Arkin, R.C., "The Multiple Dimensions of Action-Oriented Perception: Fission, Fusion, Fashion", Working notes of *AAAI 1991 Fall Symposium on Sensory Aspects of Robotic Intelligence*, Monterey, CA, Nov. 15-17, 1991.

[4] Arkin, R.C., Balch, T., Nitz, E., "Communication of Behavioral State in Multi-agent Retrieval Tasks", *Proc. 1993 IEEE International Conference on Robotics and Automation*, Atlanta, GA, 1993, Vol. 1, pp. 678.

[5] Tucker Balch, Ronald C. Arkin, "Communication in Reactive Multiagent Robotic Systems," *Autonomous Robots*, Vol. 1, 1994, pp. 1-25.

[6] Jonathan M. Cameron, Douglas C. MacKenzie, "*MissionLab*: User Manual for *MissionLab* preliminary version 0.5," *contact authors,* College of Computing, Georgia Institute of Technology, November, 1994.

[7] Brooks, R.A., "A Robust Layered Control System for a Mobile Robot", *IEEE Journal of Robotics and Automation*, Vol. 2, No. 1, March 1986, pp. 14-23.

[8] Chun, W.H., and Jochem, T.M., "Unmanned Ground Vehicle Demo II: Demonstration A", *Unmanned Systems*, Winter 1994, pp. 14-20.

[9] Gowdy, J., "SAUSAGES: Between Planning and Action", Draft Technical Report, Robotics Institute, Carnegie Mellon, 1994.

[10] Gowdy, J., "SAUSAGES: A Framework for Plan Specification, Execution, and Monitoring", SAUSAGES Users Manual, Version 1.0, Robotics Institute, Carnegie Mellon, Feb. 8, 1991.

[11] Hopcroft, J.E. and Ullman, J.D., *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, pp. 79, 1979.

[12] Kaelbling, L.P. and Rosenschein, S.J., "Action and Planning in Embedded Agents", *Robotics and Autonomous Systems*, Vol. 6, 1990, pp. 35-48. Also in *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*, P. Maes Editor, MIT Press, 1990.

[13] *Khoros: Visual Programming System and Software Development Environment for Data Processing and Visualization*, University of New Mexico.

[14] Ben Lee and A.R. Hurson, "Dataflow Architectures and Multithreading," *IEEE Computer*, August 1994, Pg 27-39.

[15] Lyons, Damian M., "Representing and Analyzing Action Plans as Networks of Concurrent Processes", *IEEE Transactions on Robotics and Automation*, Vol. 9, No. 3, June 1993, pp. 241-256.

[16] Lyons, D.M. and Arbib, M.A., "A Formal Model of Computation for Sensory-Based Robotics", *IEEE Journal of Robotics and Automation*, Vol. 5, No. 3, June 1989, pp. 280-293.

[17] Douglas C. MacKenzie and Ronald C. Arkin, "Formal Specification for Behavior-Based Mobile Robots," SPIE Mobile Robots VIII, Boston, MA, November 1993.

[18] Amol Dattatraya Mali and Amitabha Mukerjee, "Robot Behaviour Conflicts: Can Intelligence Be Modularized"," *Proceedings of the Twelfth National Conference on Artificial Intelligence*, Vol 2, Pg. 1279-1284, 1994.

[19] Saffiotti, A., Konolige, K., Ruspini, E., "A Multivalued Logic Approach to Integrating Planning and Control", Technical Report 533, SRI, SRI Artificial Intelligence Center, Menlo Park, California, 1993.

[20] K. Schwan, et. al., *A C Thread Library for Multiprocessors*, Georgia Institute of Technology Tech Report GIT-ICS-91/02, Jan. 1991.