Specification and Execution of Multiagent Missions*

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

Mobile Robot Laboratory
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280 U.S.A.

Abstract

Specifying a multiagent behavioral configuration requires both a careful choice of the behavior set and creation of a temporal chain executing the mission using those behaviors. This difficult task is simplified by applying an object-oriented approach to the design using a methodology called *temporal sequencing* to partition the mission into discrete operating states and enumerate the perceptual triggers causing transitions between those states. Several smaller independent configurations (assemblages) can then be created, each implementing one distinct operating state. Each assemblage consists of a collection of behaviors and a suitable coordination mechanism which causes the group to act as a single, coherent behavior. The missions are specified in a structured user-friendly language targeted for military-style scout missions. Various multiagent missions have been demonstrated in simulation and results are shown using our Denning mobile robots.

1 Introduction

Reactive behavior-based architectures[1, 6] 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 agents; translating those mission-oriented skills into sets of suitable behaviors (assemblages); and the construction or selection of suitable coordination mechanisms to ensure that the desired 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 does not need to 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 Georgia Tech *MissionLab* software environment has been constructed based upon the philosophy of supporting several different levels of abstraction. 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 predefined 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[7], military ter-

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

minology 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. *MissionLab* includes military map overlays, a multiagent simulator, and control windows, providing providing complete status and control.

2 Related Work

Developed at CMU, the mission specification language, SAUSAGES[8], 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 is being developed within *MissionLab* to allow targeting the UGV architecture.

Lyons' Robot Schemas (RS)[14] 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[13] 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[17]. Analysis of the correctness of configurations is necessary to support the creation of configurations by novice users. Multivalued logic techniques are expected to prove useful in such analysis.

The REX/Gapps architecture[10] 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[11] 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

MissionLab includes configuration design tools supporting the graphical construction of abstract configurations which are both robot and architecture independent. The Graphic Designer has been constructed 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 either drive the targeted vehicles or a suitable simulation.

3.1 Configuration Description Language

The context-free Configuration Description Language (CDL) provides a 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 CDL language is described in [15] and is not presented here due to space limitations.

Perceptual modules function as virtual sensors which extract features from one or more sensory 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[2]: 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. Coordinated societies of robots may also be treated as coherent assemblages, reusable in higher-level constructions.

3.2 The Configuration Network Language

When targeting the schema architecture, the CDL compiler generates a Configuration Network Language (CNL) specification of the configuration as its output. CNL is a hybrid dataflow language[12] 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.

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[18] developed at Georgia Tech. 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)[14] 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 implemented. 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.

CNL behaviors are functions which compute a single output value from multiple inputs. Figure 1 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. **goal_rel_loc** is an egocentric vector pointing towards the goal generated by another node (e.g., **detect_goal**), 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.

```
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 zero if within success circle
      VECTOR_ZERO(output);
   }
pend
```

Figure 1: Example **move_to_goal** behavior in CNL

3.3 Skill Assemblage Specification

Skill assemblages encapsulate a particular skill or ability where a skill is a higher level construction than a behavior[15]. A skill such as "follow road" may include several primitive behaviors (e.g., **stay_on_road**, **move_ahead**, and **avoid_static_obstacles**), and other skill assemblages (e.g., **avoid_dynamic_obstacles** and **maintain_formation**). These skills and behaviors coupled with suitable coordination mechanisms permit the group to function as a coherent unit while the primitive behaviors serve to ground the recursive construction.

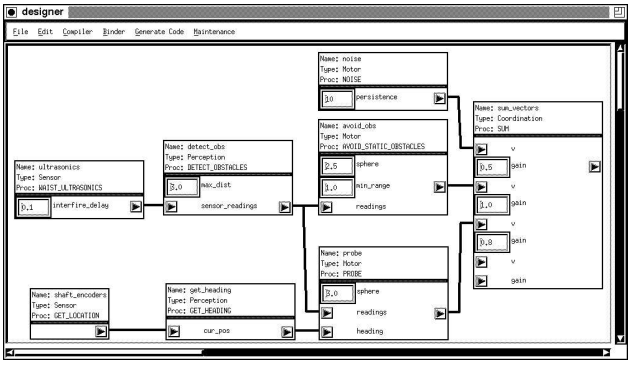

Figure 2: Parameterized wander skill assemblage

Construction of the skill assemblages by the system designer occurs using the Graphic Designer developed as part of this research. For example, Figure 2 shows an instance of a simple *wander* assemblage parame-

terized for use on our Denning robots loaded in the editor. Obstacle sensor information is provided from the ring of ultrasonic sensors on the robot, while shaft encoders maintain vehicle heading. 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.

The development process for an assemblage begins by selecting components from a library menu and placing them within the workspace. Dataflow connections are added by clicking on the corresponding input/output arrows. 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.

3.4 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 3 shows the forage assemblage[3] loaded in the graphic editor. The Wander assemblage is the component representation of the assemblage shown in Figure 2. Notice that the unconnected output of the wander assemblage in Figure 2 is the only externally visible connection in the component representation in Figure 3 (upper left component). 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 5 shows the state diagram for the forage coordination operator. Figure 4 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 mod-

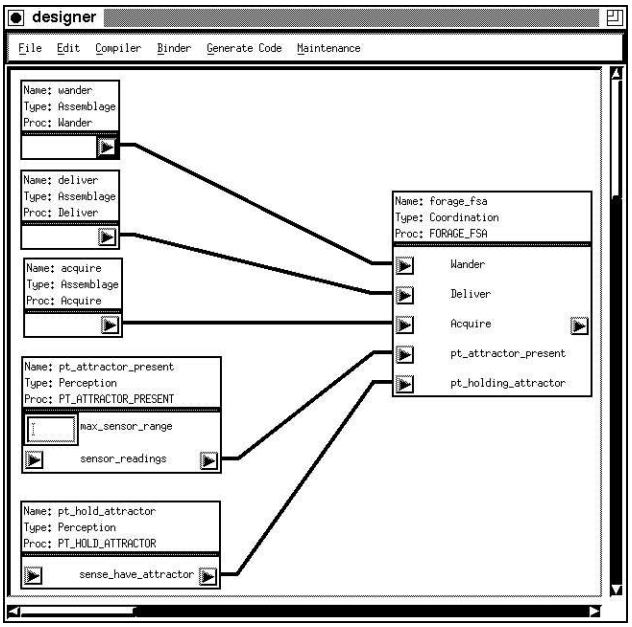

Figure 3: Abstract forage assemblage

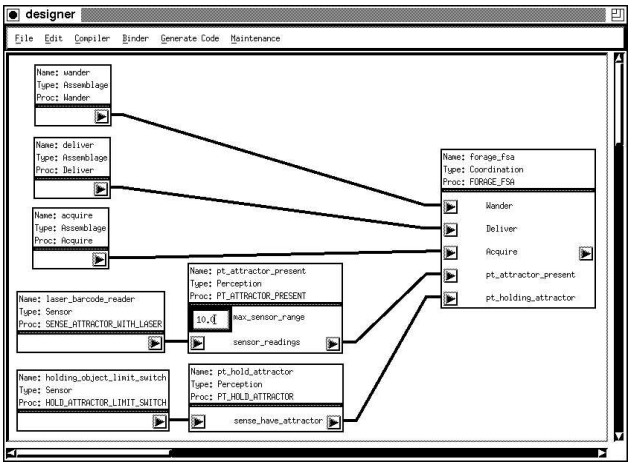

Figure 4: Parameterized forage assemblage with perceptual triggers

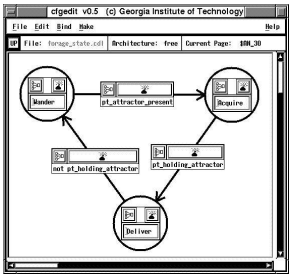

Figure 5: State Diagram for Forage Assemblage

ules. The configuration is constructed from a selected group of skill assemblages using techniques similar to those used to construct the assemblages themselves.

Figure 6 shows the complete forage configuration loaded into the graphic editor, parameterized for execution on Denning robots. The Forage assemblage is the component representation of the assemblage shown in Figure 4. The output of the assemblage will be sent to the robot for execution.

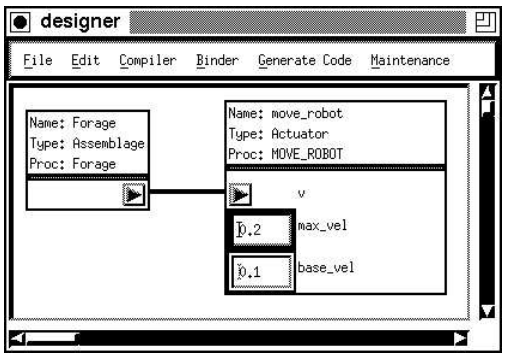

Figure 6: Forage configuration in graphic editor

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 4.1). 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 7 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

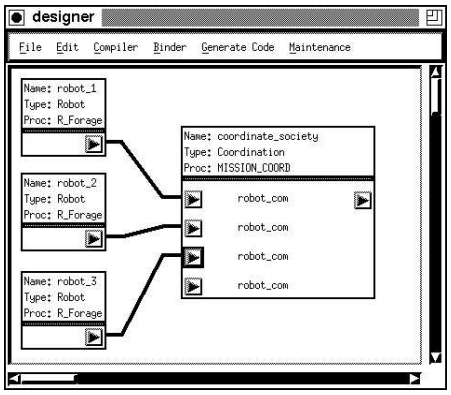

Figure 7: Configuration for three robot society

the simulation environment and the locations of all robots (simulated or real). *MissionLab* also includes other "robot programs" which simulate robots or control real robots. *MissionLab* uses "mission description" files as scripts for missions.

Figure 8 shows *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 8 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 [5].

4.1 Mission Description Language

Complex robot missions typically require construction of linear temporal chains of behaviors (e.g., moving to door, moving through door, closing door, lock door, etc.) accompanied with a multitude of contingency behaviors (e.g., if door closed open it). 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[16]. The technique of temporal sequencing[2] makes this explicit in the form of a Finite State Automaton (FSA)[9]. This provides sufficient expressive power for most missions. 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

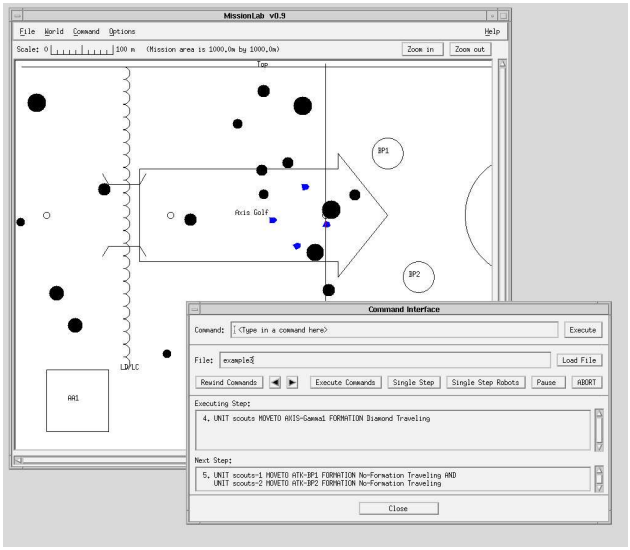

Figure 8: Example scenario in *MissionLab*

```
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 9: Example Mission Scenario Commands

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 is still running reactively. The mission coordination operator notifies the unit to activate the appropriate skill assemblage. It then waits until all the robots in the unit indicate they have 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 9. Looping and conditional constructs are not shown since they are still being developed.

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. New types of robots may also be defined and attached to an executable file name. 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 [5]. The resulting map is shown in Figure 10.

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"). Robots are drawn as black rectangles with a pointed end indicating forward.

"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
```

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 10. A timeout can also be specified for continuing automatically after a set 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.

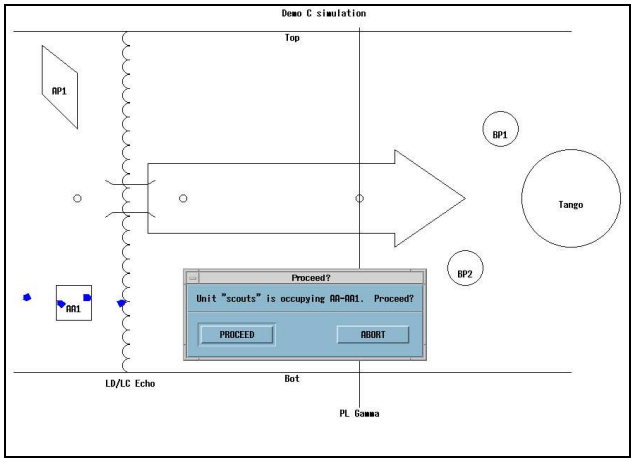

Figure 10: Mission after starting

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

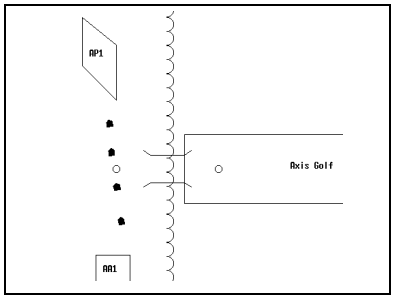

Figure 11: Unit moving to ATK-AP1

mid-execution in Figure 11. 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 12. In this case, both

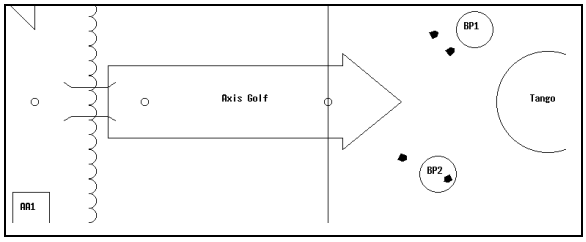

Figure 12: Unit split into subunits

subunits have `MOVETO` 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 notifies the robots in unit `scouts` that the mission is complete and terminating the control process. Further details about the command description file format can be found in [5].

5 Robotic Results

Figure 13 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, allowing use of different parameter sets with the same executable.

```
MISSION NAME "Single Denning MRV2 Robot"
OVERLAY "robot_lab.odl"
SP StartLoc 3.4 4.26
NEW-ROBOT stimpy-the-robot "robot"
  (robot_type= "MRV2",
   run_type= "SIMULATION",
   navigation_success_radius = 0.3,
   avoid_obstacle_sphere = 1.25)
UNIT <unit-stimpy> stimpy-the-robot
COMMAND LIST:
 0. UNIT unit-stimpy START StartLoc 0 20
 1. UNIT unit-stimpy MOVETO DoorWay
 2. UNIT unit-stimpy MOVETO Middle
 3. UNIT unit-stimpy MOVETO StartLoc
 4. UNIT unit-stimpy OCCUPY StartLoc
 5. UNIT unit-stimpy STOP
```

Figure 13: Script file used in missions

Figure 14 shows a screen snapshot of the mission executing in simulation with a loaded overlay representing the Georgia Tech Mobile Robot Lab. The robot is shown after the mission, having returned to its starting location in the lower left. The gap on the right is the lab door and the two passage points (shown as circles) were chosen arbitrarily as waypoints in the mission. The trail left by the robot shows it successfully completed the mission. Figure 15 shows a screen snapshot of the same mission executing on the Denning. This was create by changing the `run_type=` "`SIMULATION`" value to `run_type=` "`REAL`". 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 15). Figure 16 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. 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

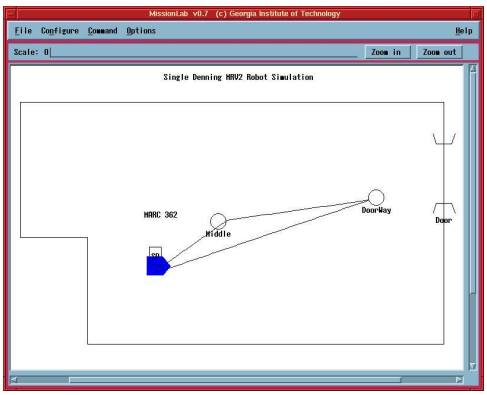

Figure 14: The mission executing in simulation

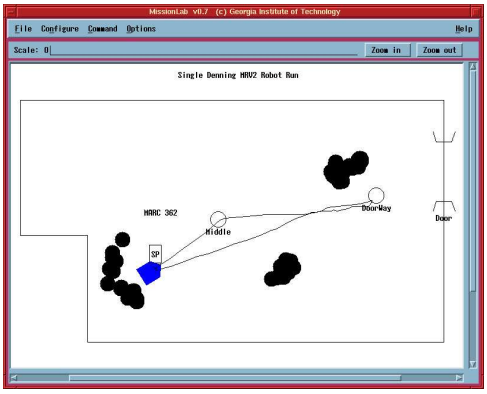

Figure 15: The mission executing on the Denning

Figure 16: Pictures of the robot executing the mission

compilers permit the construction of useful high-level behavior assemblages using a graphical interface and associated compilers that reduce 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. The *MissionLab* Web page `http://www.cc.gatech.edu/ai/robot-lab/research/MissionLab/MissionLab.html` includes user manuals and the *MissionLab* software for interested researchers.

A technology demo has been scheduled during the ARPA UGV Demo C which will highlight the systems documented in this paper. This demonstration involves operating a pair of automated off-road vehicles to accomplish several scouting tasks. This team of vehicles will 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.

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.C., "Temporal Coordination of Perceptual Algorithms for Mobile Robot Navigation", *IEEE Transactions on Robotics and Automation*, Vol 10, No. 3, June 1994, pp. 276-286.

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

[4] Balch T., and Arkin, R.C, "Communication in Reactive Multiagent Robotic Systems," *Autonomous Robots*, Vol. 1, 1994, pp. 1-25.

[5] Cameron, J.M, and MacKenzie, D.C., "*MissionLab*: User Manual", College of Computing, Georgia Institute of Technology, Nov., 1994.

[6] 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.

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

[8] Gowdy, J., "SAUSAGES: A Framework for Plan Specification, Execution, and Monitoring", Users Manual, V1.0, Robotics Inst., Carnegie Mellon, 1991.

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

[10] Kaelbling, L.P. and Rosenschein, S.J., "Action and Planning in Embedded Agents", *Robotics and Autonomous Systems*, Vol. 6, 1990, pp. 35-48.

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

[12] Lee, B. and Hurson, A., "Dataflow Architectures and Multithreading," *IEEE Comp*, Aug 1994, pp. 27-39.

[13] Lyons, D.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.

[14] 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.

[15] MacKenzie, D.C. and Arkin, R.C., "Formal Specification for Behavior-Based Mobile Robots," SPIE Mobile Robots VIII, Boston, MA, November 1993.

[16] Mali, A.D. and Mukerjee, A., "Robot Behaviour Conflicts: Can Intelligence Be Modularized", *AAAI-12 Proceedings*, Vol 2, pp. 1279-1284, 1994.

[17] Saffiotti, A., Konolige, K., Ruspini, E., "A Multivalued Logic Approach to Integrating Planning and

Control", Technical Report 533, SRI Artificial Intelligence Center, Menlo Park, California, 1993.

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