

MissionLab

User Manual for
MissionLab version 7.0

Georgia Tech Mobile Robot Laboratory

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

E-mail: mlab@cc.gatech.edu
<http://www.cc.gatech.edu/ai/robot-lab/>

July 12, 2006

About This Manual

Georgia Tech Mobile Robot Laboratory is directed by Ronald C. Arkin. This manual was created by the contributions of the following current and former Georgia Tech Mobile Robot Laboratory members (listed in alphabetical order):

Khaled S. Ali
Tucker R. Balch
Jonathan M. Cameron
Zhong Chen
Yoichiro Endo
William C. Halliburton
Michael Kaess
Zsolt Kira
James B. Lee
Douglas C. MacKenzie
Eric B. Martinson
Ernest P. Merrill
Ananth Ranganathan
Antonio Sgorbissa
Alexander Stoytchev
Patrick Ulam
Alan Wagner

This manual was compiled by Yoichiro Endo.

Copyright 1994 - 2006
Georgia Tech Research Corporation (GTRC)
Atlanta, Georgia 30332-0415
ALL RIGHTS RESERVED

This software may be modified, copied and redistributed, both within the recipient's organization and externally, subject to the following restrictions:

- (a) The recipient may not derive income for the Georgia Tech Research Corporation (herein "GTRC") software itself;
- (b) In any derivative works based on this software, the recipient agrees to acknowledge GTRC;
- (c) Any copies made of this software must be accompanied by the following copyright notice: "Copyright 1994 - 2006, Georgia Tech Research Corporation. All rights Reserved." as well as the complete copyright notice file COPYRIGHT; and
- (d) The recipient agrees to obey all U.S. Government restrictions governing redistribution or export of such information.

These restrictions may apply to redistribution within an international organization. GTRC makes no warranties or representations, either expressed or implied, with respect to the software contained herein, its quality, merchantability, performance or fitness for a particular purpose. In no event shall GTRC or its developers, directors, officers, employees or affiliates be liable for direct, incidental, indirect, special or consequential damages (including damages or loss of business profits, business interruption, loss of business information and the like) resulting from any defect in this software or its documentation or arising out of the use or inability to use this software or accompanying documentation even if GTRC, an authorized representative or a GTRC affiliate has been advised of the possibility of such damage. GTRC makes no representation or warranty regarding the results obtainable through use of this software. No oral or written information or advice given by GTRC, its dealers, distributors, agents, affiliates, developers, directors, officers or employees shall create a warranty or in any way increase the scope of this warranty. This software was developed with ARPA Grant number N00014-94-1-0215 and funded by DARPA under contract number DAAE07-98-C-LO38, DARPA / U.S. Army SMDC under contract number DASG60-99-C-0081, DARPA / DOI contract number NBCH1020012, and NAVAIR Intelligent Autonomy program.

Contents

1	What is <i>MissionLab</i>?	1
1.1	<i>MissionLab</i> Overview	1
1.2	<i>MissionLab</i> Development History	2
2	Getting Started with <i>MissionLab</i>	4
2.1	Installing <i>MissionLab</i>	4
2.2	Running Quick Example Programs	6
2.2.1	Demo-1 (<i>mlab</i>)	6
2.2.2	Demo-2 (<i>CfgEdit</i>)	7
3	The <i>MissionLab</i> Tools	16
3.1	<i>mlab</i> (User Interface Console)	16
3.1.1	Staring <i>mlab</i>	16
3.1.2	<i>mlab</i> Command Line Arguments	17
3.1.3	X-Motif Resources for <i>mlab</i>	19
3.1.4	File Utilities	19
3.1.5	Obstacle Creation Dialog Box	19
3.1.6	World Scale Dialog Box	20
3.1.7	Time Scale Dialog Box	20
3.1.8	Dynamically Changing the Environment	21
3.1.9	Command Interface Panel	22
3.1.10	CMDLi Interface Panel	24
3.1.11	Sound Simulation	24
3.1.12	Motivational Vector Window	25
3.1.13	<i>Telop</i> Interface	25
3.1.14	Personality Window	27
3.1.15	Robot Trails	29
3.1.16	Obstacle Highlighting	29
3.1.17	Movement Vector	29
3.1.18	Movement Field	29
3.1.19	Sensor Readings Display	30
3.1.20	Data Logging	31
3.1.21	Debug Options	31

3.1.22	Show Line Of Sight	32
3.1.23	Compass Window	32
3.1.24	3D Display	33
3.1.25	Help Menu	33
3.2	Robot Executables	34
3.3	<i>CfgEdit</i> (Configuration Editor)	35
3.3.1	Reviewing the Configuration File	35
3.3.2	Running <i>CfgEdit</i>	39
3.3.3	Browsing the Configuration Tree	40
3.3.4	Building a Mission	44
3.3.5	Using Waypoints Tool	44
3.3.6	Using Path Planning Tool	55
3.3.7	Using Q-Learning for a Behavioral Selection	61
3.3.8	Analyzing and Compiling the Robot Executable	64
3.3.9	Executing the Robot Executable	64
3.3.10	Adding New States and Triggers	67
3.3.11	Using Mission Expert	70
3.4	CDL (Configuration Description Language)	72
3.5	CNL (Configuration Network Language)	77
3.6	CMDL (Command Description Language)	81
3.6.1	Mission background information part	81
3.6.2	Mission Command List Part	85
3.6.3	Example Command Description File	88
3.7	ODL (Overlay Description Language)	89
3.7.1	Scenario Information Part	89
3.7.2	Control Measure Description Part	90
3.7.3	Example Overlay Description File	92
3.8	<i>HServer</i> (Hardware Server)	94
3.8.1	Configuration File	94
3.8.2	Command Line Arguments	97
3.8.3	Environment Variables	98
3.8.4	User Interface and Key Commands	99
3.9	<i>CBRServer</i> and CBR Mission-Planning Wizard (“Mission Expert”)	104
3.9.1	Overview	104

3.9.2	Invoking <i>CBRServer</i>	104
3.9.3	Creating an Example Mission Using Mission Expert	106
3.9.4	“Expert” Menu in <i>CfgEdit</i>	111
3.9.5	Repairing a Faulty Mission Using the Playback Function	111
3.9.6	CBR-CNP Integration	112
3.9.7	Customizing Preference and Constraint Sets and Toolbox	113
4	Support Software	121
4.1	Communications Software IPT	121
4.2	Process Control Software C-Threads	121
5	Current Limitations of <i>MissionLab</i>	122
6	FAQ / Trouble Shooting	123
7	Feedback and Bug Reports	124

1 What is *MissionLab*?

1.1 *MissionLab* Overview

MissionLab is a powerful set of software tools for developing and testing behaviors for single robots and a group of robots. Code generated by *MissionLab* can directly control commercial robots. ATRV-Jr / Urban Robot (iRobot), AmigoBot / Pioneer AT / Pioneer 2DX (ActivMedia, Inc.), and Nomad-150 / 200 (Nomadic Technologies, Inc.) are among those robots *MissionLab* has supported successfully. A primary strength of *MissionLab* is its support of both simulated and real robots. A developer can experiment with behaviors in simulation and then run those same configurations on mobile robots (Figure 1).

MissionLab has a distributed architecture. Thus, the main user's console can run on one computer while multiple robot control executables are distributed across a network, potentially on-board the actual robots they control.

The core of the *MissionLab* tool-set is composed of six primary components:

- *mlab*: *mlab* is a console-like program from which a user monitors the progress of experimental runs of the robot executables. Locations of the robots and detected obstacles are examples of various data *mlab* can monitor. When *mlab* is used for simulation (as opposed to controlling mobile robots), it serves as a sensor and actuator simulator from the point of view of the robot executable. On mobile robots, the actual sensors are used instead. Details of *mlab* are described in Section 3.1.
- *CfgEdit*: The Configuration Editor, or *CfgEdit*, is a graphical tool for building robot behaviors. The designer can build complex control structures with the point and click of a mouse. *CfgEdit* generates source code which, when compiled, can directly control a simulated or real robot. Details on *CfgEdit* are provided in Section 3.3.
- *cdl*: The *cdl* code generator translates the CDL (Configuration Description Language), which is generated by *CfgEdit*, into CNL (Configuration Network Language) code. In general, users will not need to be concerned with CNL. However, because programming in CNL is very similar to programming in the C language, advanced users may develop their own primitive behaviors and store them as a library and/or write their own control programs without using *CfgEdit*. CDL is explained in Section 3.4, and CNL is explained in Section 3.5. However, for details on CNL, please read the separate CNL manual.
- *cnl*: The *cnl* compiler compiles CNL code generated by the *cdl* code generator, and produces C++ code. Once this C++ code is compiled with the GNU C Compiler (*gcc*), the compiled program (or robot executable) may now directly control a robot. The *cnl* compiler is automatically invoked by *CfgEdit* when needed.
- *HServer*: *HServer* (Hardware Server) directly controls all the robot hardware, either via TCP/IP or a serial link, and provides a standard interface for all the robots and sensors. The *CfgEdit* generated code uses this standard interface to control the real robots. *HServer* also provides direct control, configuration, and status of the robots and sensors. Details on *HServer* are described in Section 3.8.
- *CBServer*: *CBServer* (CaseBased Reasoning Server) generates a mission plan based on specs provided by the user by retrieving and assembling components of previously stored successful mission plans. Details on *CBServer* are described in Section 3.9.

In addition to CDL and CNL described above, there are two more original languages that were specifically developed for the *MissionLab* system:

- **CMDL**: The Command Description Language (CMDL) may optionally be used for describing simple sequential robot missions. A CMDL file, containing both background and command information, will

be read by *mlab* at runtime and offer a mechanism for providing high-level input to robot behaviors developed in CNL. When robot executables are directly created by *CfgEdit*, users may not need to use CMDL. CMDL is described in Section 3.6.

- **ODL:** The Overlay Description Language (ODL) provides descriptions of the environment, which *mlab* can graphically translate to a map or a floor plan of an experimental area. For example, a robot's starting point, obstacles, boundaries, are among those features that ODL can describe. ODL is described later in this manual (Section 3.7).

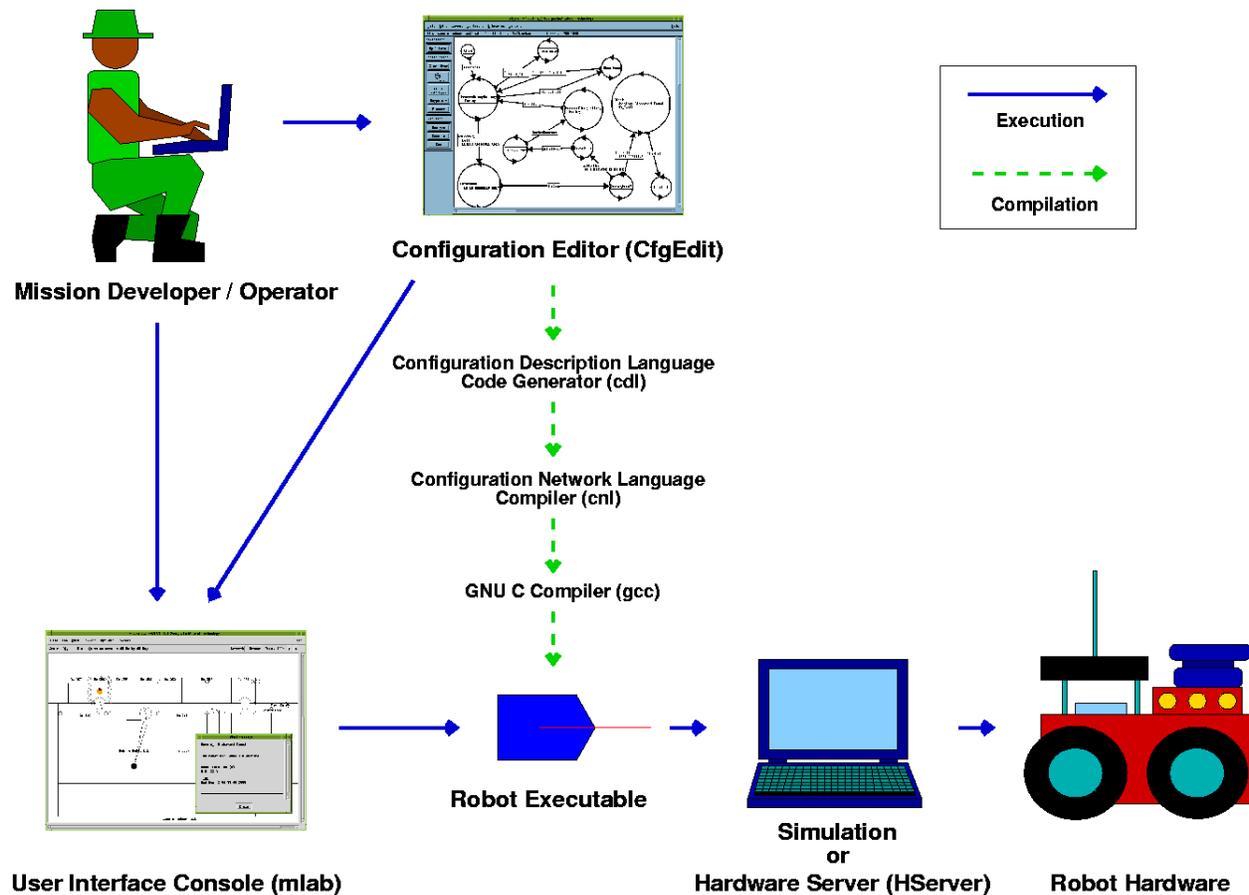


Figure 1: *MissionLab* Components: the mission developer can create a mission for a robot or a group of robots with the Configuration Editor (*CfgEdit*). Once the mission is created, it can be compiled as a robot executable, which will give commands to the robot hardware or its embedded low-level software via Hardware Server (*HServer*). The robot executable is executed by the User Interface Console (*mlab*), and *mlab* can be invoked by both *CfgEdit* and the operator. *mlab* can also run a simulation before the actual real robot run.

1.2 *MissionLab* Development History

MissionLab is a product of ongoing work in the College of Computing at Georgia Institute of Technology (Georgia Tech), which addresses several aspects of configuring and controlling teams of robots, particularly in realistic military scenarios. This work has been initially conducted under the Defense Advanced Research Projects Agency (DARPA) task “Flexible Reactive Control for Multi-Agent Robotic Systems in Hostile

Environments” (DARPA Task #A447). The main goal for this project was to control the motion of a robot or groups of robots in highly dynamic, unpredictable, and possibly hostile environments. Much of our work in reactive control was applicable to suitably controlling the motion of individual robots [2] as well as the motion of teams of robots [6].

When developing software to control teams of robots prior to *MissionLab*, it became apparent that manually determining each configuration of robots and their control software, hard-coding a task, and controlling task execution was complex and error-prone. Thus, part of our research aimed towards the reduction of this complexity by developing tools for configuring robots and their individual control software [14]. We began to focus on the ability to specify the instructions for a task in such a manner that computer software can interpret them and decide which commands to send to each robot (simulated or real). As a result, the *MissionLab* structure of five primary components *mLab*, *CfgEdit*, *cdl*, and *cnl* was developed, and they were included in earlier *MissionLab* versions.

Since the release of *MissionLab* version 3.0, we have switched the target operation system from SunOS to Linux. For version 4.0, a number of robot behaviors were added for urban warfare operations (DARPA TMR), and a new tool, *HServer*, which can control various robot and sensor hardware, was created.

In DARPA MARS project, the focus of our research was addition of a framework to *MissionLab* that enables robots to synthesize the desirable features and capabilities of both deliberative (symbol-mediated) and reactive (sensor-mediated) control. The research involved: (1) Planning with incomplete information; (2) The use of case-based reasoning (CBR) methods for situation-dependent behavioral gain and assemblage switching at the reactive level of execution; (3) The use of CBR for reasoning in Finite State Automata (FSA) plan generation through the use of “wizards” to guide high level deliberative planning; (4) Specialized reinforcement learning methods (“learning momentum”) to adjust behavior gains at run-time; and (5) The integration of Q-learning methods for behavioral assemblage selection at a level above gain adjustment.

In DARPA MARS-2020 project, we have conducted studies on integrating *MissionLab* into a larger robotic system involving coordination of the robots that are launched from different software platforms such as University of Pennsylvania’s *ROCI* and University of Southern California’s *Player/Stage*. Mobile Intelligence Corporation’s *CMDLi* was integrated into *MissionLab* in order to coordinate the communication among those heterogeneous robots. Furthermore, communication sensitive behaviors (communication recovery, communication preservation, and internalized plan) were developed for this project, and *HServer*’s localization capability was improved by incorporating probabilistic sensor fusion methods (Extended Kalman Filter, Particle Filter, etc.).

In ongoing NAVAIR Intelligent Autonomy project, we are improving the mission specification process by combining the CBR Wizard with an auction-based task allocation algorithm (Contract-Net Protocol).

2 Getting Started with *MissionLab*

2.1 Installing *MissionLab*

The following explains the procedure for how to install *MissionLab* on your machine:

1. **Ensure you have an appropriate operating system.** We have successfully compiled and tested *MissionLab* with the following operating systems.

- **Red Hat Linux 9.0**
- **Fedora Core 3, 4, and 5**
- **Red Hat Enterprise Linux 4 WS**

Note 1: The default installation of a RedHat Linux system may not include all of the packages necessary to compile *MissionLab*. Upon installing Linux, we recommend you to select all the packages labeled under *Development Tools*, *X Window System*, *X Software Development*, and *Legacy Software Support*.

Note 2: If you installed Fedora Core 5 straight from the installation disks, *MissionLab* may not compile due to the problem in the kernel (2.6.15). In this case, we recommend you to update your kernel to at least 2.6.17 and other software packages by running *yum*. If you are new to Linux and not familiar with how to update the kernel, we recommend you to use Fedora Core 4 instead.

Note 3: The most common problem for compiling *MissionLab* is due to the fact the GNU compiler (*gcc/g++*) is too new. The newest GNU compiler that *MissionLab* is known to compile with is *gcc 4.1* (Fedora Core 5). Please be aware that any compiler whose version is higher than 4.1 may fail to compile *MissionLab*.

Note 4: We noticed that LessTif version of Motif (API for X Window GUI) tends to produce various warnings during the operation of *MissionLab*, and it even fails to show some of the widgets if executed from Fedora Core 5. Hence, we recommend you to use Open Motif version of Motif.

Note 5: *MissionLab* requires a 3-button mouse to operate. If you do not have a 3-button mouse, make sure that emulation of 3-button is enabled.

2. **Make sure you have an enough disk space to install *MissionLab*** (300 MB minimum; 400 MB recommended).
3. **Obtain the latest *MissionLab* distribution** (if you haven't yet). You can download the latest distribution of *MissionLab* (*MissionLab-7.0.tar.gz*) from our web page:

<http://www.cc.gatech.edu/ai/robot-lab/research/MissionLab/>

Note: Supporting Software is included with the *MissionLab* distribution. You are not required to take any special steps regarding this software, but we list it here to credit the developers and point out copyright restrictions:

- **IPT** - IPT is an interprocess communication system developed at Carnegie Mellon University. Also see the file *README.IPT*. This release of the *MissionLab* was developed with version 8.4 of IPT. If you wish, you can obtain the package from: <ftp://ftp.cs.cmu.edu/user/jayg> (download *ipt-8-4.tar.gz* and *IPTManual.ps.gz*). Due to copyright restrictions, IPT MAY NOT BE USED FOR COMMERCIAL PURPOSES.
- **CThreads** - CThreads is a lightweight process threads package from Georgia Tech which operates under Linux, SunOS and many other OSs. If you wish, you can download the package from: <http://www.cc.gatech.edu/systems/projects/Cthreads>

- **CMDLi** - Developed by Mobile Intelligence Corporation, CMDLi is a communication package that coordinates heterogeneous robotic software platforms. For example, a mission involving robots launched from Georgia Tech's MissionLab (Linux), University of Pennsylvania's ROCI (Windows), and University of Southern California's Player/Stage (Linux) can be simultaneously coordinated via CMDLi. For more details on CMDLi, contact Mobile Intelligence Corporation directly (<http://www.mobile-intelligence.com>).

4. **Unpack the *MissionLab* distribution.** Uncompress and untar the distribution with the following command:

```
tar -xvzf MissionLab-7.0.tar.gz
```

You should see a directory named `MissionLab-7.0`. This directory will be referred to as the “MissionLab home” directory.

5. **Compile:** Compile *MissionLab* with the following commands:

```
cd [your MissionLab home]/src
./configure
make all
```

Note: The `configure` script, which calls `makedepend`, may generate numerous warnings, but they can be safely ignored.

6. **Set up your user environment.** After compilation, modify your `.tcshrc`, etc. to setup your environment:

- Add “[*your MissionLab home*]/bin” into your execution path.
- Add “.” into your execution path.

At this point, the installation is completed, and you should be able to run the *MissionLab* system.

2.2 Running Quick Example Programs

To check if your installation of *MissionLab* has been properly compiled, you can run several demo programs that are included in the *MissionLab* package. Here are two of those example programs that might help you to understand how *mlab* and *CfgEdit* work in the *MissionLab* system.

2.2.1 Demo-1 (*mlab*)

Run *demo_marc* and you should be able to observe what the *mlab* (*MissionLab* User Interface Console) looks like. Type the following commands (Note: “marc” stands for Manufacturing Research Center at Georgia Tech, where our Mobile Robot Laboratory used to be):

```
cd [your MissionLab home]/demos/tmr_demos
which mlab - Make sure [your MissionLab home]/bin is in your execution path
demo_marc
```

At this point, if your screen pops up the *mlab* console (Figure 2), and the robots start moving, *MissionLab* is running successfully. To quit the program, click “File” in the menu bar, and select “Quit”. See Section 3.1 for more details on how to run *mlab*.

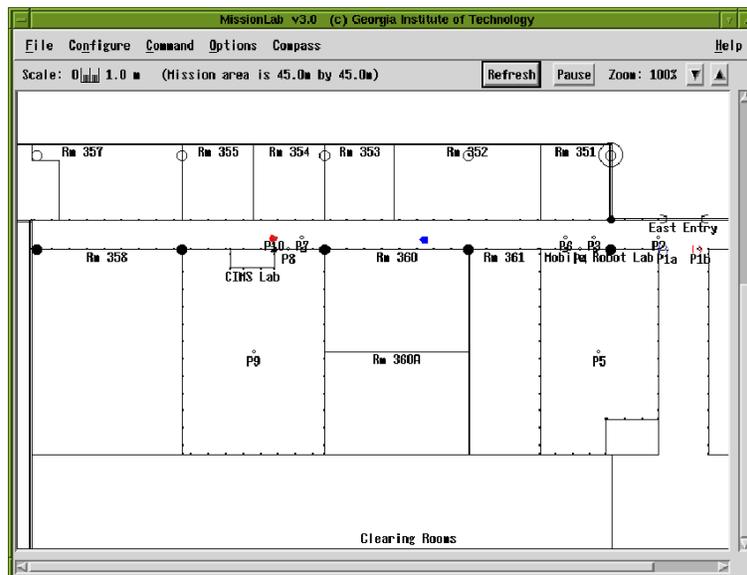


Figure 2: Example scenario being executed in *MissionLab*

2.2.2 Demo-2 (*CfgEdit*)

At this point, you may also want to check to see if *CfgEdit* works properly in your new installation. Follow these steps to create a simple “back_and_forth”-robot using *CfgEdit* :

1. Go to an appropriate directory

- In the directory you are going to run *CfgEdit*, make sure that there is “Empty.ovl”. You need this to load the default overlay when you invoke *mlab* from *CfgEdit*.

```
% cd [your MissionLab home]/demos/mars_demos
% ls Empty.ovl
```

2. Run iptserver

- You must be running *iptserver* when the *CfgEdit* invokes *mlab*. Read Section 3.1 for the explanation.

```
% iptserver &
```

3. Run CfgEdit

- If you launch *CfgEdit* with the following command, its window (Figure 3) will pop up on the screen.

```
% cfgedit
```

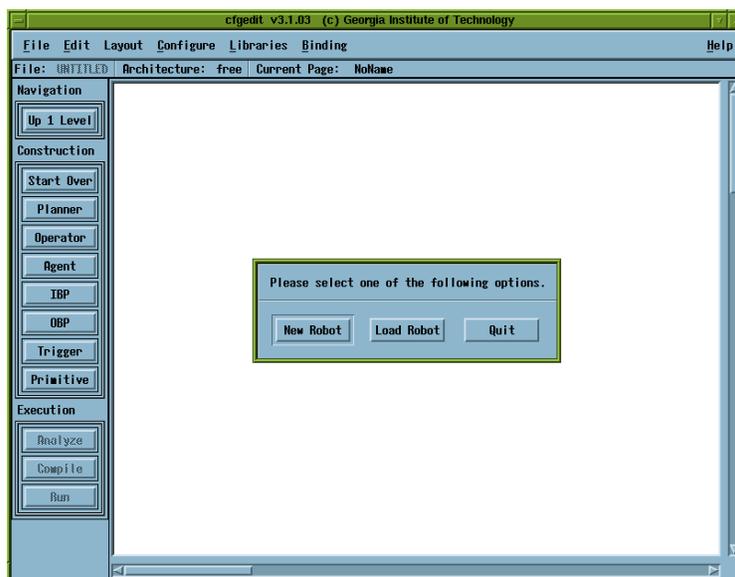
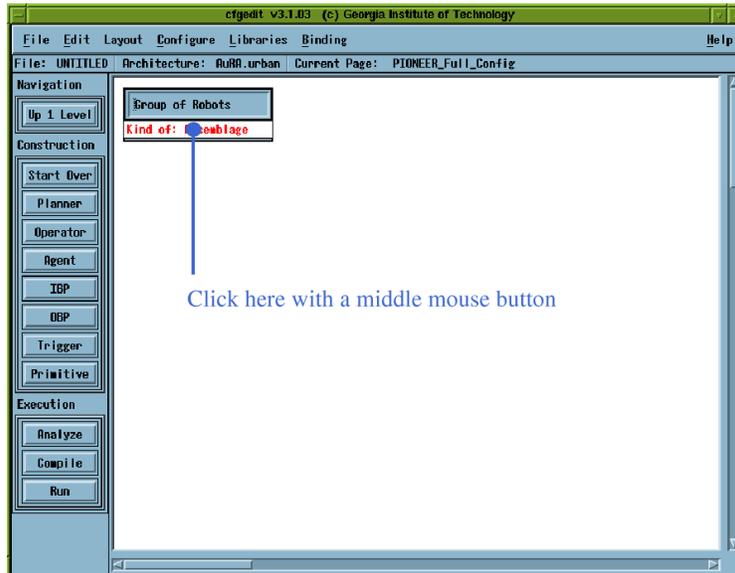


Figure 3: *CfgEdit* Window

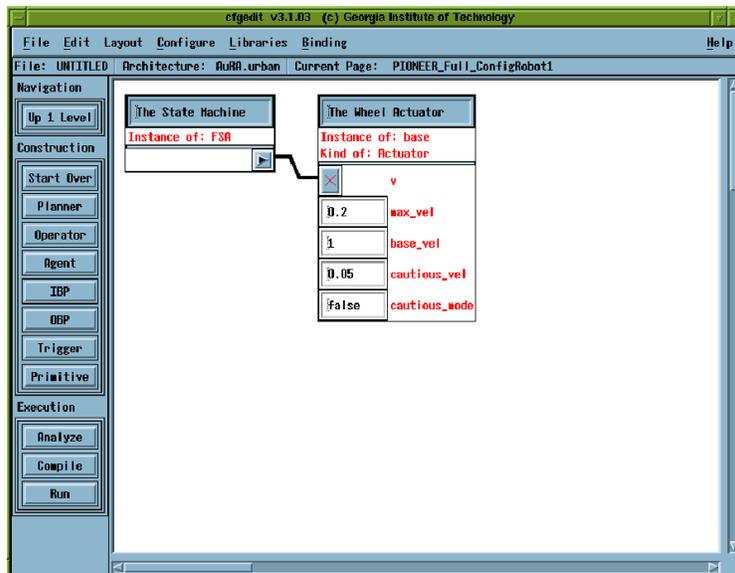
4. Move to a FSA diagram

- Choosing “New Robot” from the first selection. If your *MissionLab* was installed without the CBR Mission Planning Wizard functionality (Mission Expert), *CfgEdit* will show the top level of the

Figure 4: *CfgEdit* - the top level

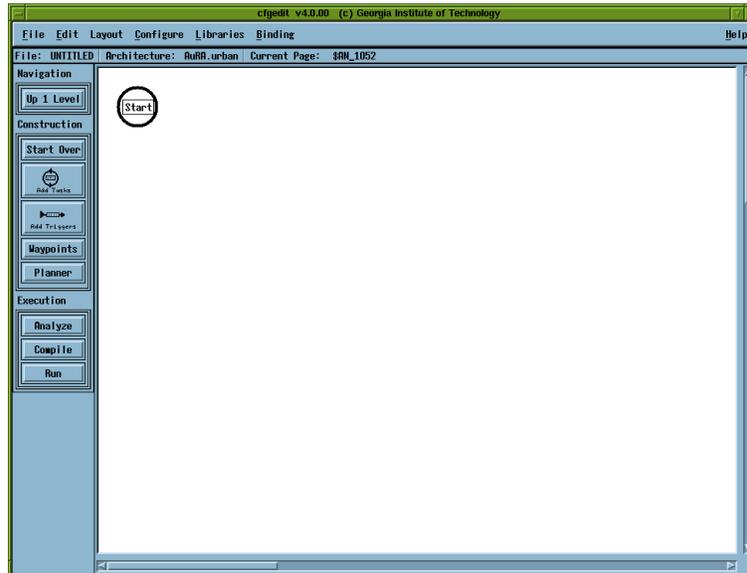
configuration as shown in Figure 4. If Mission Expert is installed, *CfgEdit* asks you whether to use it. Select “No” to the question, and you will see the top level of the configuration.

- Click the box labeled with “Kind of: Assemblage” under the heading “Group of Robots” with the middle button of the mouse (middle-click).
- If you middle-click again in the box for “Instance of: DEFAULT_ROBOT”, you will see the display like the one in Figure 5.

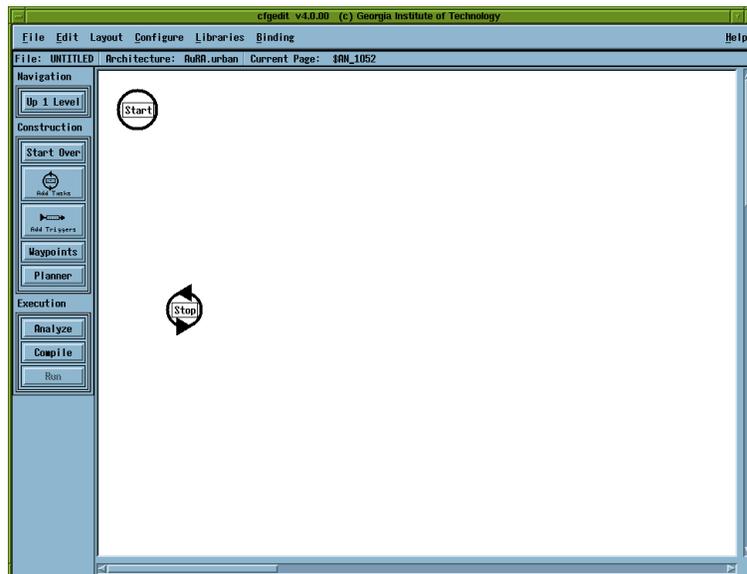
Figure 5: *CfgEdit* - the third level

- By middle-clicking the box for “Instance of: FSA” under “The State Machine”, you can reach the Finite State Acceptor (FSA) diagram window (Figure 6).

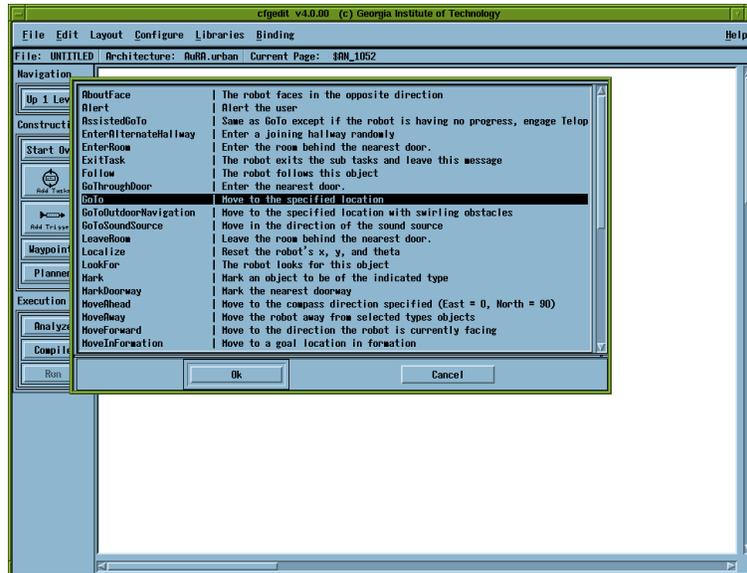
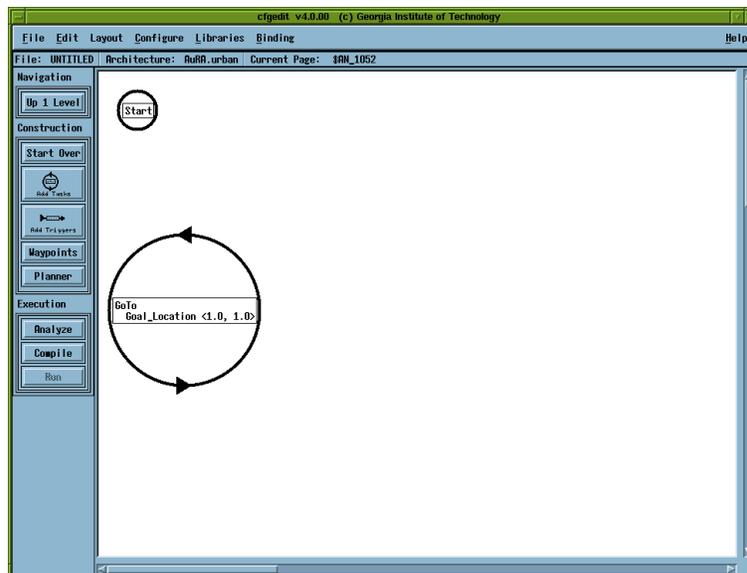
5. Constructing a Back_and_Forth Robot

Figure 6: *CfgEdit* - the FSA diagram

- At this moment, you should see a circle labeled “Start” (Figure 6). This circle indicates the robot’s “start” state. Now, you will add its first “GoTo” state by left-clicking the **Add Tasks** button located in the left menu bar.
- Move the pointer into the display area, and left-click again. You will see a new state whose default is “Stop” (Figure 7).

Figure 7: *CfgEdit* - adding a new state

- In order to modify the state into another state, right-click the “Stop” state, and choose “GoTo” (left-click) from the list (Figure 8).
- Make sure you have created the new “GoTo” state as it is shown in Figure 9, and change the goal location by middle-clicking the state.

Figure 8: *CfgEdit* - modifying the stateFigure 9: *CfgEdit* - the “GoTo” state added

- Type in a goal location (in the world coordinate), for example, to be $X = 10$, and $Y = 20$ (Figure 10).

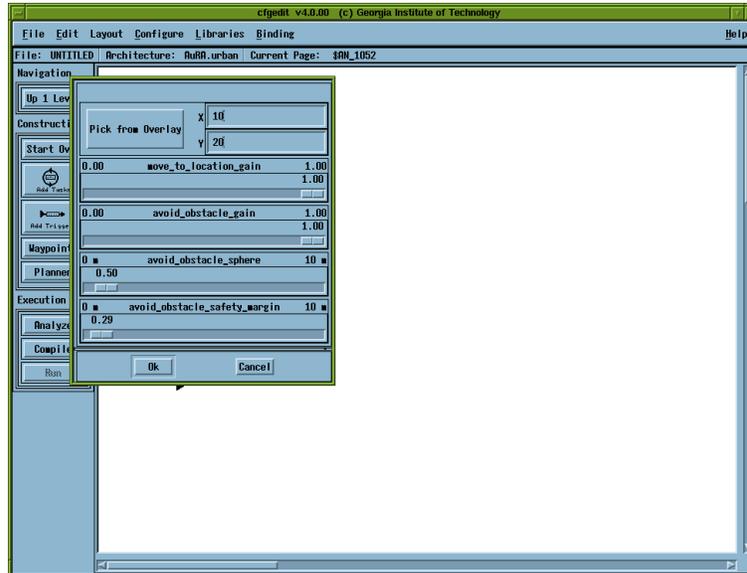


Figure 10: *CfgEdit* - specifying a goal location

- Now, in order for the robot to have a transition from one state to another, you need to add a trigger. Left-click the **Add Triggers** button located in the left menu bar, press down the left button of the mouse in the “Start” state, drag the arrow to the “GoTo” state, and release it. You will find a trigger connected between those two states (Figure 11). If you click the **Add Triggers** button by mistake, you can cancel this process by selecting “Cancel” in the “Layout” menu.
- By repeating the procedures above, add another “GoTo” state ($X = 30$ and $Y = 20$), and connect the first “GoTo” state to this one with a trigger.
- The trigger connected between the first “GoTo” state and the second “GoTo” state should have automatically appeared as “AtGoal”, and the value for the goal location should be automatically copied from the first “GoTo” state ($\langle 10, 20 \rangle$). Even though this is correct and you do not have to modify this trigger, you can right-click on it to see the rest of the triggers that are available (Figure 12). If you want to modify the parameters for the trigger, you can middle-click on it like you did for the state (Figure 13).
- By adding one more trigger from the second “GoTo” state to the first one with the at-goal location to be $\langle 30, 20 \rangle$, you have completed creating a back_and_forth robot (Figure 14).

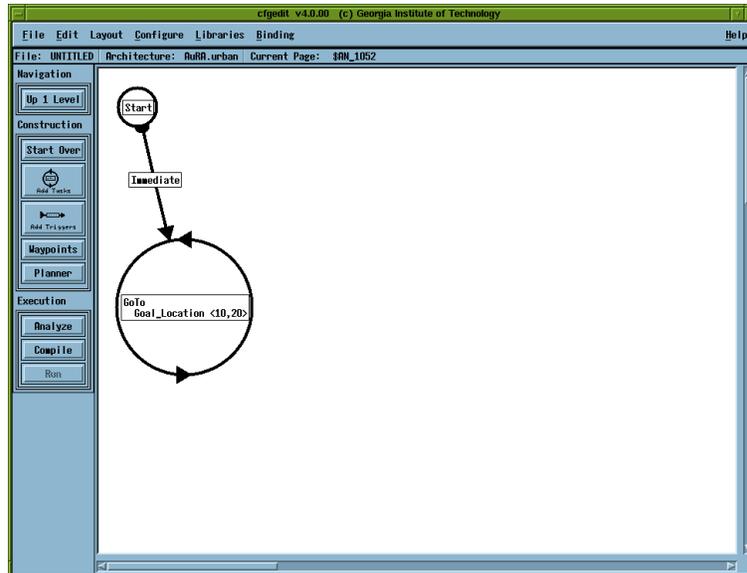
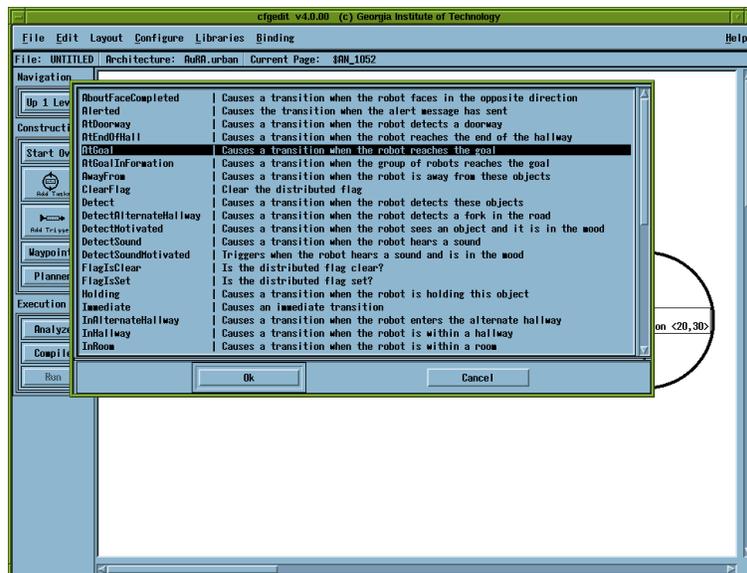
6. Save the Back_and_Forth Robot Configuration

- You can save this configuration for its future use by selecting “Save Configuration **A**s” from the “File” menu on top of the window.

7. Compiling the Robot Executable

- After creating the configuration for the back_and_forth robot, you can now create a robot executable by left-clicking the “Compile” button in the left menu bar.
- You will see a pop-up window (Figure 15) showing compilation progress.
- Left-click **OK** when the compilation is finished.

8. Running the Back_and_Forth Robot

Figure 11: *CfgEdit* - a trigger addedFigure 12: *CfgEdit* - a list of triggers can be seen by right-clicking on one of the triggers you created.

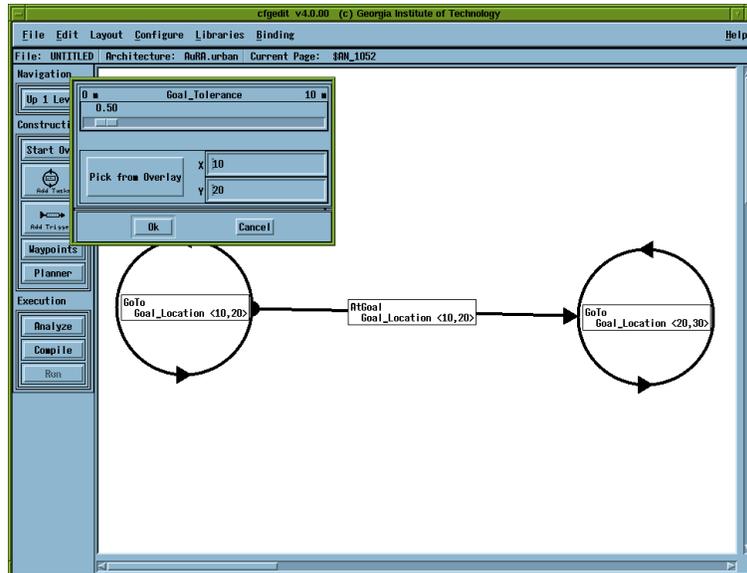


Figure 13: *CfgEdit* - the parameters for a trigger can be modified by middle-clicking on the trigger.

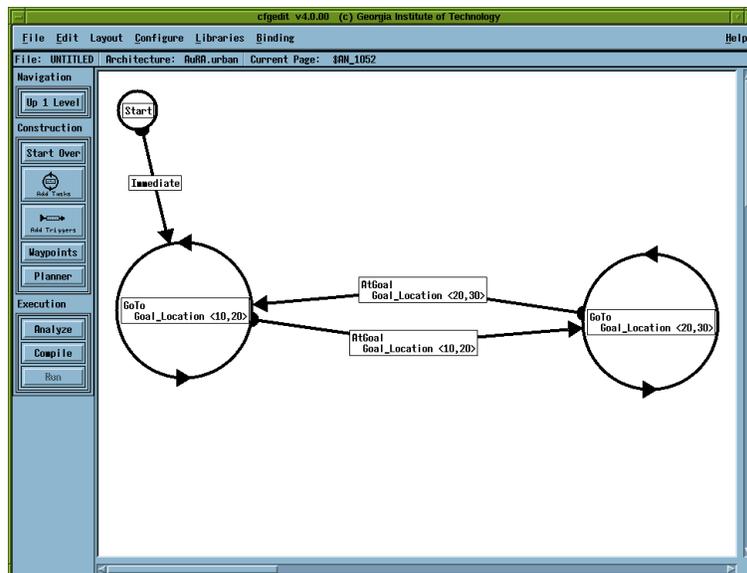
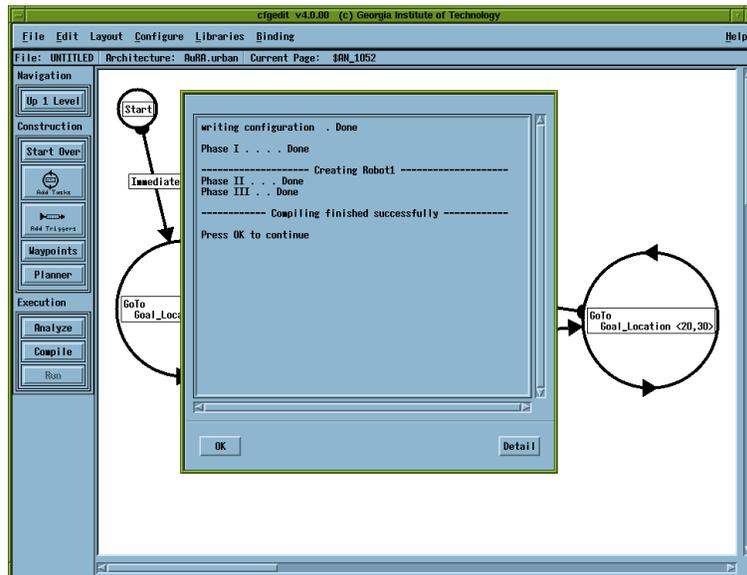
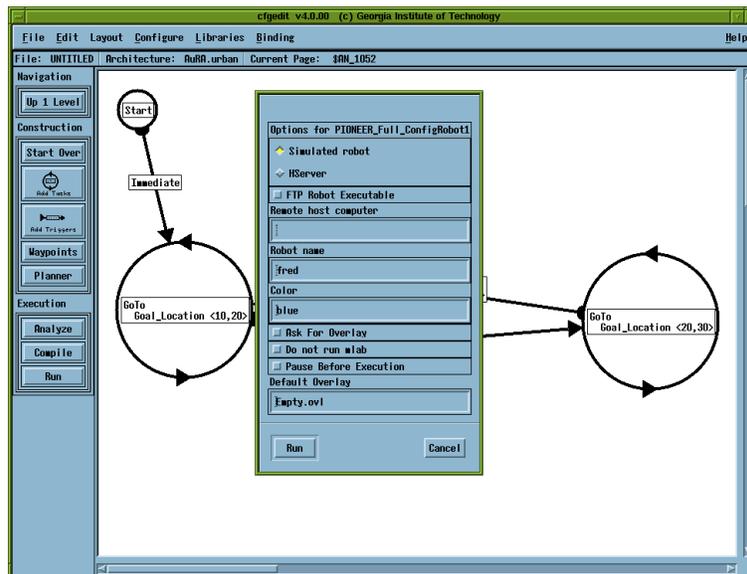


Figure 14: *CfgEdit* - the back_and_forth robot configuration

Figure 15: *CfgEdit* - compilation

- Finally, you can run this robot you just created. Left-click the “Run” button in the left menu bar, and select “Simulated robot” (default) and left-click the **Run** button (Figure 16).
- If the *mlab* console shows a robot going back and forth between two points, like the one in Figure 17, congratulations, you have successfully created the robot executable using *CfgEdit*!

Figure 16: *CfgEdit* - running *mlab*

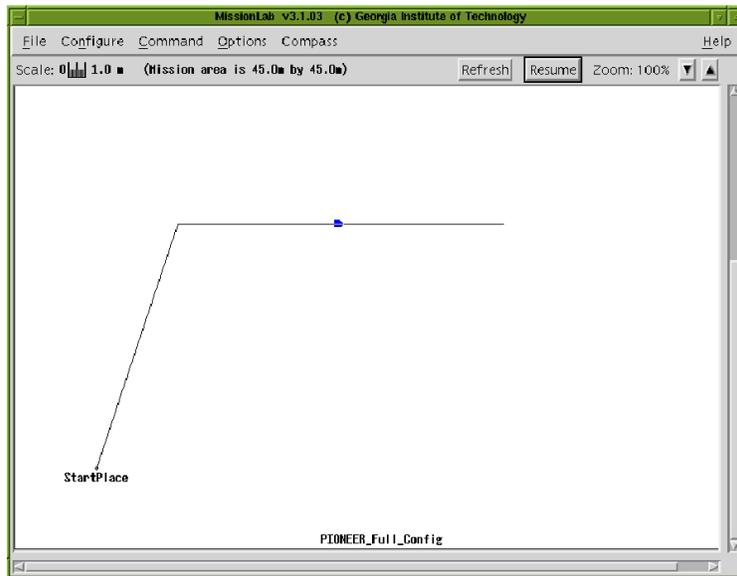


Figure 17: *mlab* - running the back_and_forth robot

3 The *MissionLab* Tools

3.1 *mlab* (User Interface Console)

3.1.1 Staring *mlab*

When you ran the quick demo program in the previous section, you ran a script called “demo” in the directory to execute *mlab*. However, to be able to utilize *MissionLab* functions, you need to be able to run *mlab* manually. Take the following steps to run *mlab*:

1. First, the communications server, IPT or `iptserver`, must be operating. If `iptserver` is not already running on an accessible computer, run the version supplied with the *MissionLab* package in the background. You should be able to find `iptserver` in `[your MissionLab home]/bin` directory where you should already have the execution path pointing to:

```
% iptserver &
```

If you see an error like “IPT: Failure to create module listen socket.”, most likely, it means that `iptserver` is already running on the machine. You can safely ignore the error.

It might be advantageous to run `iptserver` in a separate window since it generates output which does not usually need to be monitored. If an `iptserver` process is already running, note the host that it is running on. If it is the host on which *MissionLab* is to be run, no further preparation is necessary. If not, there are two ways to inform *MissionLab* where to locate the server. The first is via an environment variable, `IPTHOST`. Suppose the communication server is running on host “cartman.cc.gatech.edu”. The following command will set the necessary environment variable:

```
% setenv IPTHOST cartman.cc.gatech.edu
```

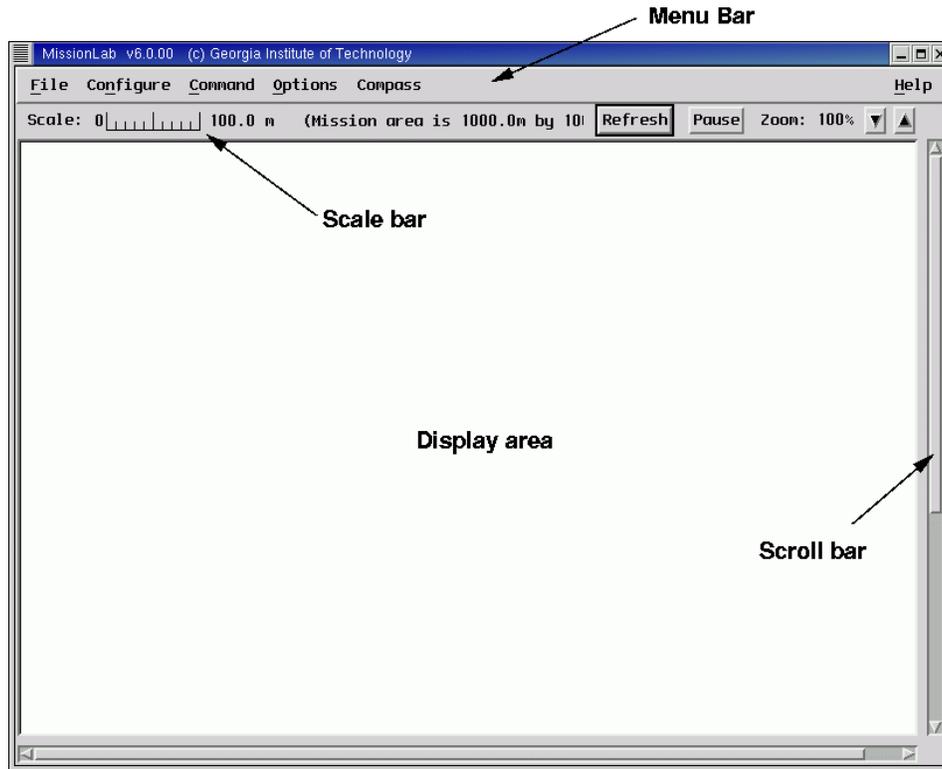
It may not be necessary to spell out the complete internet name of the host in this command. The second way to specify the communications server host is via a command line argument, which is explained later.

2. *mlab* is now ready to be run. Move to the appropriate directory and bring up the user interface console by running *mlab*:

```
% mlab
```

At this point, the user interface for displaying the scenario and controlling its operation should be popped up (Figure 18). This user interface console is called *mlab*, and it consists of the following four components:

- **Menu bar:** The menu bar contains the menus for “File”, “Configure”, “Command”, “Options”, and “Compass”.
- **Scale bar:** The scale bar is underneath the menu bar. It shows the scale by displaying a length legend. It also has a display of the current value of the zoom factor, the percent the display is scaled up from its nominal size. On the right of the zoom factor value are two buttons:  (Zoom out) and  (Zoom in) which can be used to zoom the map display area out and in. For example, when the display area is filled with detected obstacles, you can press `Refresh` to clear them all. The `Pause` button will allow you to stop the movement of a robot for both simulation and real-robot cases.

Figure 18: *mlab* - MissionLab User Interface Console

- **Display area:** The largest part of the user interface is the display area in its center. The operational area is shown in this area.
- **Scroll bars:** On the lower right and bottom edges of the user interface are the scroll bars for scrolling around the display area.

In this section, the usage of *mlab* is explained.

3.1.2 *mlab* Command Line Arguments

mlab can take several optional arguments. The full form of the form is:

```
mlab [X options] [Other options] [CMDL file] [-n]
```

All of the arguments are optional. **X options** is a normal X options such as *-geometry*, etc. The main argument, **CMDL file**, is the name of a command description (CMDL) file to load as part of the startup process. Other options are:

- 3 Enable the 3D mode.
- A Enable the assistant dialog.
- b Enable the playback mode. Pick your own robot log data.
- B Repair mission with the playback mode. Run with -M.
- c *filename* Use a specified configuration file instead of *.cfgeditrc*.

-
- C Show the command panel on start up.
 - d Enable debugging.
 - D *filename* Display the CMDLi panel. Use the specified CMDLi file.
 - e *filename* Enable event logging. Use the specified file.
 - h Show usage.
 - H *height* Set the height of the main window.
 - I *hostname* Use the specified address as the IPT host.
 - l Show color map.
 - L Log the robot data.
 - m *filename* Pick a coordinate of a single point on the map and write in the. outfile.
 - M *string* Enable the Mission Design mode. The specified 'string' is to name output files.
 - N *cnpmode* Set the CNP mode. The input can be either a number or string:
 - 0 or CNP_MODE_DISABLED
 - 1 or CNP_MODE_PREMISSION
 - 2 or CNP_MODE_RUNTIME
 - 3 or CNP_MODE_PREMISSION_AND_RUNTIME
 - p *filename* Use the specified personality file.
 - P Pause before execution.
 - r Start the mission as soon as *mlab* is loaded.
 - R Pick the overlay of the mission manually.
 - s *number* Set the random number seed.
 - S Display the current state information during the execution.
 - t *number* Time to expire *mlab* (in second).
 - T Show the telop interface on start up.
 - w *filename* Enable the waypoint mode and output the coordinates in the file.
 - W *width* Set the width of the main window.
 - X Show the fallback resources.
 - n Do not show the copyright dialog.

3.1.3 X-Motif Resources for *mlab*

MissionLab operates best on color monitors but should work adequately on monochrome monitors. *mlab* uses the X-Motif widget set but should run under any X-based window manager. The defaults for various X-Motif related display characteristics can be overridden by the user by adding resource statements to their X11 `.Xdefaults` file. For instance, the following resource statement may be useful:

```
MissionLab*background: LightSkyBlue3
```

This directive causes all the backgrounds in *MissionLab* to be displayed using the “LightSkyBlue3” color. The size of the main display window can be changed by using the following geometry directive:

```
MissionLab.geometry: 850x600+10+200
```

(This same directive could have been given on the command line also using the `-geometry` option. Note that this directive does not affect the size of the scrollable map area.) To change the characteristics of the command interface panel, use the resource `MissionLab*Command Interface.*`. For the obstacle creation dialog box, use `MissionLab*Obstacle Creation.*`. For the world scale dialog box, use `MissionLab*World Scale.*`. For the time scale dialog box, use `MissionLab*Time Scale.*`. For a few other resource names, run `mlab` using the `-X` option to see the fallback resources.

3.1.4 File Utilities

As shown in Figure 19, the “File” menu has four entries. The first entry, “Open Mission”, invokes a file

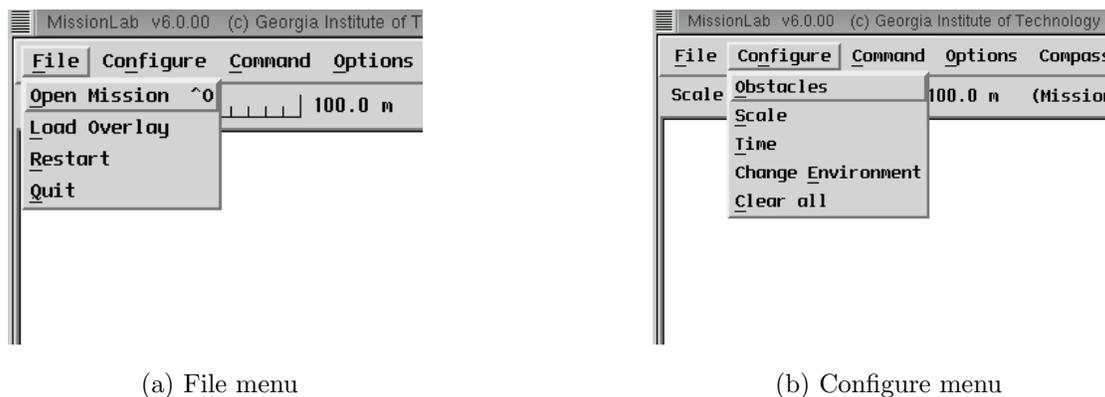


Figure 19: The file and configure menus

dialog box which can be used to locate and load command description files (CMDL). The details of CMDL are described in Section 3.6. You can also use a shortcut key `Ctrl-o` to invoke this function. The second entry, “Load Overlay”, is for viewing an overlay file (ODL). ODL is used to describe a mission area of robots you are going to experiment with. The details of ODL are explained in Section 3.7. The third entry, “Restart” rewinds a mission when it is finished, and runs it again. When you want to kill the *MissionLab* process, you can choose “Quit” from this menu.

3.1.5 Obstacle Creation Dialog Box

The first entry of the `Configure` menu (Figure 19) is `Obstacles`, and it invokes the obstacle creation dialog box (Figure 20). This dialog box allows the user to populate the world with obstacles. Thus, when you are

using *MissionLab* for simulating and experimenting, for example with an obstacle avoidance behavior, you might find this function to be useful.

The parts of the dialog box are as follows: The three buttons, **Ok**, **Apply**, and **Cancel**, along the bottom have typical meanings. An additional button, **Clear Obstacles**, allows the operator to delete all the obstacles. When the parameters are satisfactory, press the **Ok** button or the **Apply** button to generate a new field of obstacles. The **Ok** button hides the dialog box while the **Apply** button leaves it up for further operations. The **Cancel** button hides the dialog box without generating a new set of obstacles. The top scale (**Obstacle Coverage**) controls what percent of the area is to be covered by obstacles. Note that obstacles generated by this dialog box will not overlap. The second and third scales from the top control the minimum and maximum size of the obstacles (in meters). The text field labeled “Seed:” is the current value of the seed for the random number generator. The user can enter a value into that field and tell the system to accept it by pressing the **Accept** button or by pressing RETURN-key with the focus in the seed text field. The user can have the system generate a new random seed by pressing the **Generate new seed** button. The user can also choose to have the software generate a new seed every time the **Apply** button is pressed by turning on the “Auto Randomize” toggle button.

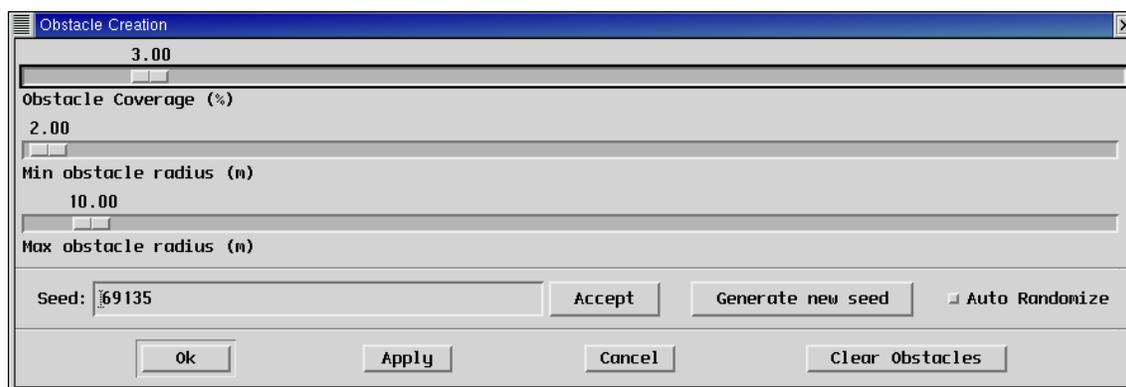


Figure 20: The obstacle creation dialog box

3.1.6 World Scale Dialog Box

The second entry of the **Configure** menu is **Scale**, which invokes the world scale dialog box (Figure 21). The world scale dialog box allows the user to alter the scale of the display of the world. The **Ok**, **Apply**, and **Cancel** buttons have their typical meanings. The scale at the top controls the zoom factor used to display the map area. This value is also controlled by the  (zoom in) and  (zoom out) buttons just below the right end of the menu bar. The “Scale Robots” toggle controls whether the size of the displayed robots is to be scaled with the world. If it is on, the robots are shown as being as long (in meters) as the “Robot Length” scale on the right indicates. If it is off, the robots are shown at a fixed size (regardless of the zoom factor); specifically, the robots are shown as being as long (in pixels) as the “Robot Length” scale on the right indicates.

3.1.7 Time Scale Dialog Box

The time scale dialog box (Figure 22) can be invoked by the third entry of the **Configure** menu, **Time**. The time scale dialog box allows the user to alter the speed at which the simulation executes. The scale of the duration of each robot sense/compute/act cycle, in seconds can be adjusted using the slider bar. Note that this controls the simulated time taken per step, not the actual CPU time required to execute it.

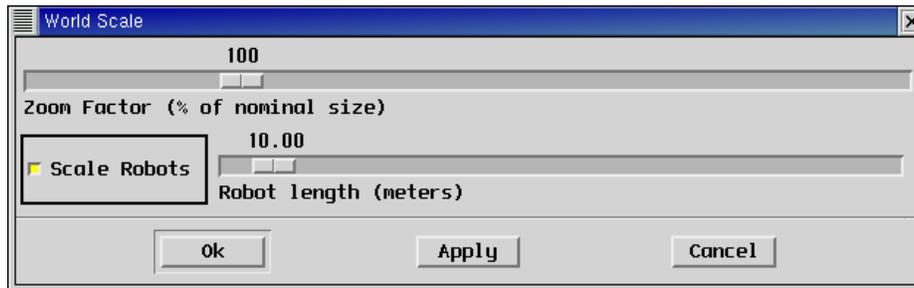


Figure 21: The world scale dialog box

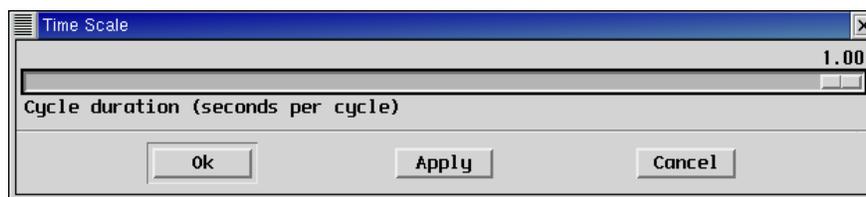


Figure 22: The time scale dialog box

3.1.8 Dynamically Changing the Environment

MissionLab allows for dynamically changing the simulation environment via an “environment file” that describes what changes to make. The file can either be specified by the `-E` option on the command line or via the “Change Environment” option on the **Configure** menu before the mission has begun if *mlab* was started in the paused state.

An example of environmental file “sample_environmental_file” can be found in “demos/mars_demos”. An environment file should consist of a list of the following sections:

```
STEP_COUNT count [period]
a list of actions
```

where *count* is the step count at which *a list of actions* should be executed. *period* is an optional whole number. If it’s present (say it’s value is *P*), then the list of actions will be carried out again *P* steps into the future, and again *P* steps after that, and so on. So, the list of actions corresponding to this section will be carried out every $(count + P \cdot x)$ steps, where *x* is a whole number.

a list of actions is one or more lines (the `STEP_COUNT` section cannot be empty), where each line specifies a different action to be carried out. Right now, obstacles can be created or destroyed either individually or in groups. The currently available rules follow:

1. To add a single obstacle at a given position with a given radius:

```
ADD OBSTACLE x y radius
```

2. To remove an obstacle at a given position:

```
REMOVE OBSTACLE x y
```

3. To add a group of obstacles randomly distributed within a region:

```
REGION x1 y1 x2 y2 ADD OBSTACLE coverage min_rad max_rad [seed[,seed inc]]
```

where $(x1, y1)$ and $(x2, y2)$ define the lower left and upper right corners of a rectangle. This rectangle defines the region to be populated with obstacles. *coverage* is a number between 0 and 100 that defines the percentage of the region to be covered. *min_rad* and *max_rad* are the min obstacle radius and the max obstacle radius, respectively, to use. All obstacle radii will be uniformly distributed between these two numbers. *seed* is an integer used to seed the random number generator. It's optional, but include it if you want to reproduce the same obstacles later. Specifying a *seed* value, however, cannot totally guarantee two identical obstacle fields if a robot is present inside the region. Obstacles will not be created over robots, so it is possible for the behavior of the robot to affect the creation of obstacles. If this is a periodic action (i.e. you create obstacles, then destroy them, then re-create them, and so on...), then you may also want to add the optional *seed inc* parameter. This is an integer value which is used to increment the random seed every time this action is carried out.

Some examples are:

```
REGION 0 0 10 10 ADD OBSTACLE 10 0.5 1.5
REGION 0 0 10 10 ADD OBSTACLE 20 0.5 1.5 5
REGION 0 0 10 10 ADD OBSTACLE 30 0.5 1.5 5,2
```

All of these will fill regions within the rectangle defined by (0,0) and (10,10). The first line specifies 10% coverage, the second specifies 20%, and the third specifies 30%. All lines specify obstacles with radii between 0.5 and 1.5 meters. The first doesn't specify a seed value, and the second specifies a value of 5. The third line specifies that the seed value should initially be 5, but in subsequent executions of this line, the seed should be incremented by 2 each time, so seeds for this line will be 5, 7, 9, and so on.

4. To delete a group of obstacles within a region:

```
REGION x1 y1 x2 y2 REMOVE OBSTACLE
```

As above, the region is the rectangle defined by $(x1, y1)$ and $(x2, y2)$. All obstacles within the rectangle will be deleted.

3.1.9 Command Interface Panel

The first entry of the Command menu (Figure 23) is Command Interface, and it is used to bring up the command interface panel, shown in Figure 24.

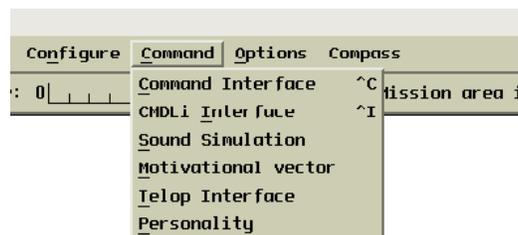


Figure 23: The command menu

The command interface panel can be hidden at any time by pressing the **Close** button on the bottom of the panel. The top area of the command interface panel is a text field labeled “Command:” in which the user can enter commands manually. The command can be executed by pressing the **Execute** button on the right side of the text field or by pressing RETURN while the focus is in the text field. The second area

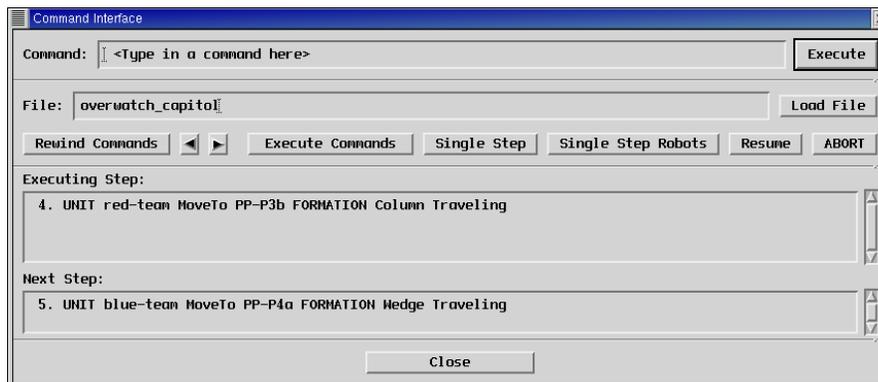


Figure 24: Command Interface

from the top is the command list execution area. It has a text field labeled “File:” in which the user can enter names for command files (Section 3.6) to load. The file is loaded when the **Load File** button on the right is pressed or when the RETURN key is pressed while the focus is in the “File:” text field. The functions of the buttons in the lower part of this area are explained below.

The third area from the top is the command display area. The top part of it is where executing steps are displayed. The lower part shows the next command to be executed. If the set of commands in the step display is larger than the display window, the user can browse through it using the scroll bars on the right.

The control buttons in the bottom part of the command list execution area have the following functions: If the **Execute Commands** button is pressed, the list of commands is automatically executed. This is the primary mode of operation. A secondary mode of operation is to single step through the list of commands using the **Single Step** buttons. The rest of the buttons have the following functions:

Rewind Commands Rewind the list of commands. This cannot be invoked if commands are executing automatically.

 Move backwards through the list of steps. If commands are executing automatically, the command list backs up and the new command is executed. If the commands are not executing automatically and a single step is executing, the command list backs up but the new command is not executed. If no commands are executing, the command list backs up without executing the new command.

 Similar to , except that the command list is moved forward.

Execute Commands Execute the list of commands automatically, starting with the command shown in the “Next Step:” display window. Note that the first command is usually a “START” start command. It is an error to start with other commands, such as “MOVETO” without starting the robots first.

Single Step Execute the next command in the command list. (If commands are executing automatically, this is similar to the  button.)

Single Step Robots When the robots are paused, they can be single-stepped by using this button. Each press of this button causes the robots to go through one sense-move cycle.

Pause Pause the motion of the robots. The button then changes to **Resume**. Press it again to let the robots resume their motion.

ABORT Abort the execution of the current list of commands, delete the robots, and rewind the command list.

3.1.10 CMDLi Interface Panel

CMDLi was developed by Mobile Intelligence Corporation in order to enhance CMDL's capability for coordinating multiple heterogeneous robots that are launched from multiple heterogeneous software platforms (Georgia Tech's *MissionLab*, University of Pennsylvania's *ROCI* and University of Southern California's *Player/Stage*). The interface itself (Figure 25) is similar to the command interface panel described in the previous section. In both cases, a mission is written in a script-like language (CMDL). However, the difference is that, unlike the previous command interface where *mlab* reads the instructions from the script and control robots by acting as a central command, in CMDLi, every robot has its own copy of the CMDL script, and carries out the integrated mission in a distributed fashion. If necessary, the robots can be synchronized by CMDLi.

There are two example CMDLi missions provided in the *MissionLab* packages. Follow the procedure below to execute those missions:

- **To run the first demo - All robots are launched from a single *mlab* process**
 1. `cd [your MissionLab home]/demos/mars2020_demos`
 2. `./run-mini-demo`

- **To run the second demo - Multiple robots are launched from three different *mlab* processes**
 1. `cd [your MissionLab home]/demos/mars2020_demos`
 2. `cfgedit gtech.cdl`
 3. compile, but do not run
 4. `quit CfgEdit`
 5. `cfgedit non-gtech.cdl`
 6. compile, but do not run
 7. `quit CfgEdit`
 8. `./CopyDemoRobots`
 9. `./run-big-demo` (Press OK to proceed when asked)

3.1.11 Sound Simulation

MissionLab has the capability to simulate sound sources. The interface to simulate sound sources can be invoked from the **Command** menu and is called "Sound Simulation." It appears similar to the *telop* interface (Section 3.1.13). To simulate a sound, the user is required to click inside the white disk labeled `Sound.Source` (Figure 26). After the click, a line is drawn connecting the center of the disk with the click location. The direction of this line corresponds to the direction of the sound source and its length corresponds to the sound volume.

This interface generates a continuous and sustained sound (in simulation). To stop the sound, the user should middle click inside the white circle, or left click again to specify a different sound (direction and volume). To quit the sound simulation click the **End SoundSim** button.

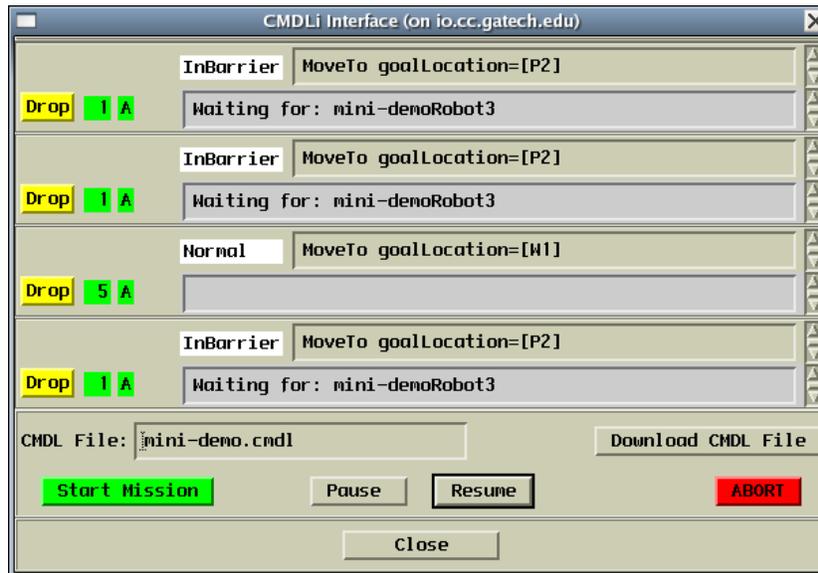


Figure 25: CMDLi Interface

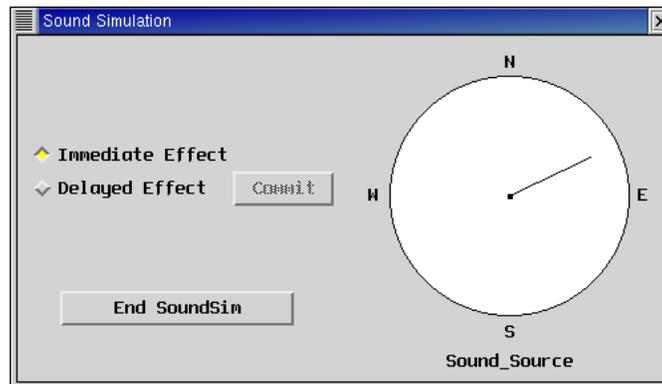


Figure 26: Sound Simulation Window

3.1.12 Motivational Vector Window

To provide a variety of behaviors, *MissionLab* was augmented with motivational variables. These variables correspond to internal states of the robot and can be used to simulate motivations and/or emotions. These variables are kept in a database and their values can be updated by different states and triggers or they can be directly changed by the user through the use of the motivational vector interface (Figure 27). This interface can be invoked from the **C**ommand menu. Please note that this is similar to the Personality Interface (Section 3.1.14) but implemented differently. For example, this interface can only apply to the *CfgEdit*-type robot while the personality function is only implemented in the manually-coded-type-robot. (See Section 3.2 which explains the difference between these two types of robots.) The behaviors that utilize this motivational variables are listed in Section 3.3.4.

3.1.13 *Telop* Interface

In some cases, you may want to drive a robot using a mouse or joystick to influence the direction in which the robot moves. Some robots you create with *CfgEdit* or a generic robot executable that come with the

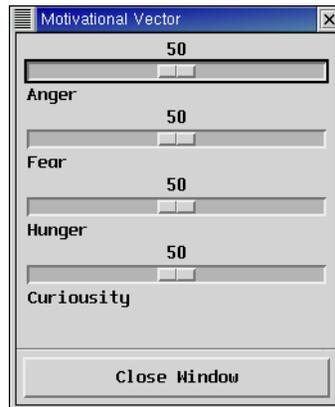
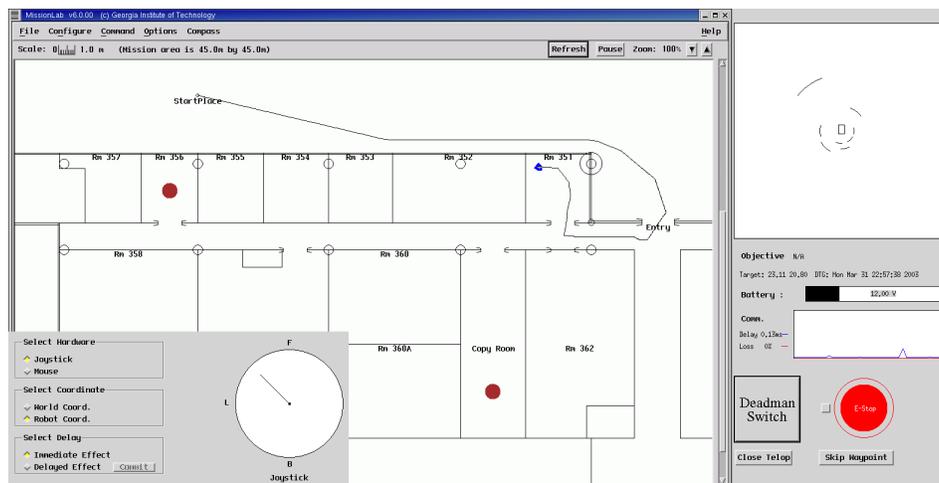


Figure 27: Motivational Vector Window

MissionLab package are implemented with *Telop* capability, which allow users to maneuver robots with a joystick-like interface (Figure 28). The Teleautonomous Operation (*Telop*) interface can be invoked by choosing the fourth entry (Telop Interface) of Command menu.

Figure 28: *Telop* Interface

The *Telop* interface is divided into two sections. The section on the upper right is the *Status Window*. On the upper right of the *Status Window* there is an egocentric view of the sensor data. The small rectangle represents the robot, facing forward (towards the top of the screen). This is exactly the same as the sensor display in the main *MissionLab* window except it is from the robot's perspective. Below that is the *Objective* display which shows the world coordinates of the current goal of the robot. The first two numbers are the coordinates of the goal, the next is the distance to the goal, and the last is the angular distance to the goal (L is for left and R is for right). Under the *Objective* is the *Target* display, which shows the current coordinates of the robot itself, along with the current date and time. The *Battery* display shows the current voltage of the battery on the robot, if such a reading is available. The *Comm* display shows the communication between the robot and the host computer, both in terms of ping loss and ping delay (in milliseconds). The **E-Stop** button is the emergency stop button which will halt the robot if it is pressed. Next to it is the **Deadman Switch**, which does nothing if clicked with the mouse but will be explained later in the context of using a real joystick. The **Skip Waypoint** button will cause the `AtOrSkipGoal` trigger to become true and cause a transition in the FSA, if such a trigger was used. Finally, the **Close Telop** button will remove the *telop* interface and end teleoperation.

On the bottom there is an on-screen “joystick”, which appears as a white circle with a dot in the center. On the left of the “joystick” are controls that affect the use of the *Telop* interface. The Immediate Effect and Delayed Effect toggle buttons control when an input to the “joystick” is seen by the robots. In Immediate Effect mode, any input is seen by the robots immediately. In Delayed Effect mode, the robots do not act on inputs until the `Commit` button is pressed. Also, the *Select Coordinates* toggle buttons allow the user to choose whether the vector added from teleoperation is in the robot’s local coordinate system (with the robot at the origin and the axes rotated) or in the world coordinate system.

Note that there are two ways that the *Telop* interface can be used. First, if a command list is executing, the *Telop* interface can be popped up at any time (via the menus) to override or direct the motion of all the robots. Second, if the user actually embeds a TELEOPERATE command in the list of commands, then the *Telop* interface will pop up automatically and will only control the specified unit. See the TELEOPERATE command on page 87 for details of the command syntax involved. Either of these two events will cause the telop interface to pop up and resize the main MissionLab window to fit within the telop framework. Once the interface appears the user can resize the main window, or if done with the teleoperation the user may click the `Close Telop` button and the telop interface will disappear. The user may also move around the two sections of the telop interface by pressing the ALT key while simultaneously clicking on the desired window and dragging it.

Finally, there are two devices which can be used with the teleoperation interface: a mouse or a real joystick. The *Select Hardware* toggle buttons allow the user to choose which input device is being used.

Using a Mouse

To depress the “joystick” in a particular direction, move the mouse pointer to a position in the “joystick” and click the left mouse button. A line is drawn from the center of the “joystick” to the position clicked to indicate the direction and amount that the “joystick” is depressed. Notice in the figure that a line is drawn to the northwest. This commands the robots to move towards the northwest. The “joystick” remains depressed until the user clicks in the “joystick” with the middle mouse button, and then the “joystick” will return to the neutral position.

Using a Real Joystick

In order to teleoperate the robot with a real USB joystick (connected via `/dev/js0`), the main trigger button of the joystick must be pressed. The `Deadman Switch` button will reflect whether the trigger button is pressed properly or not. If it is not, movement of the joystick will *not* result in any change in movement of the robot. If the trigger is pressed, then moving the joystick to the desired direction will result in teleoperation and will be reflected in the display as a line in the “joystick” display.

3.1.14 Personality Window

The last entry in the `Command` menu is `Personality`, which invokes the personality window (Figure 29). The personality window allows the user to interactively modify the behavioral parameters values of the robots¹ in terms of the individual parameters or more abstract groupings such as personality traits.

This personality window contains two slider-bars that affect the aggressiveness and wanderlust of the robots. Moving the slider-bar to the right increases the trait, while moving the slider-bar to the left decreases the trait. Increasing the aggressiveness increases the robots’ desire to reach their goal location while decreasing the desire of robots to avoid obstacles. Increasing the wanderlust increases the amount of noise in the robots’ movements while decreasing their desire to stay in formation.

The user can specify his own set of personality traits by creating a personality file and specifying it on the command line with the `-p` option. An example personality file, called *personality.file*, can be found in `[your MissionLab home]/demos/mlab-2.0.demos/` directory. The format of the personality file is a single line

¹Currently, the personality function is only implemented in the manually-coded-type-robots (*mlab-2.0-robot*, *tmr-robot-1998*), which are explained in Section 3.2. Thus, it does not apply for the *CfgEdit*-type robot.

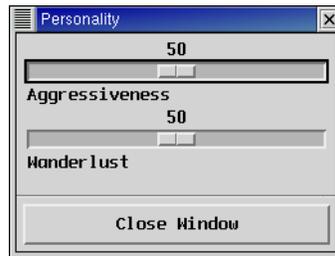


Figure 29: Personality Window

that indicates the number of personality sliders followed by groups of information for each personality. The group for each personality contains a title for the slider-bar, as well as information regarding each of the behavioral parameters that the slider-bar effects. The title must be a single word. Each behavioral parameter is specified by the key for the behavioral parameter (as it appears in the robots' databases), a base value for the parameter, and an indication of whether the parameter is to be incremented or decremented when the personality trait is increased.

It is easiest to explain the format of the personality file by examining the example file.

```
Number of sliders: 2
```

```
title: Aggressiveness
num_params: 2
key: navigation_move_to_goal_gain
base: 0.8
inc: 1
key: navigation_avoid_obstacle_gain
base: 1.5
inc: 0
```

```
title: Wanderlust
num_params: 2
key: navigation_noise_gain
base: 0.1
inc: 1
key: navigation_formation_gain
base: 1.0
inc: 0
```

The first entry indicates that there are two personality traits in the window. Then for each personality trait, there is a group of lines. The first group is for the aggressiveness trait. The `title:` line indicates that the personality trait is to be labeled Aggressiveness. The `num_params:` line indicates that this is a grouping of two parameters. Each of these parameters is then specified by a `key:`, `base:`, and `inc:` line. The `key:` line indicates that the first of the two behavioral parameters affected by this slider-bar is identified in the robots' database by the key `navigation_move_to_goal_gain`. The `base:` line indicates that the base value for this behavioral parameter is 0.8. Each parameter can be adjusted in the range from 0 to two times the base value. When the personality slider is at the 50% value, then the behavioral parameter will be at the base value. The `inc:` line indicates that the first behavioral parameter will be increased as the personality trait is increased. The value of 1 in this line indicates that the behavioral parameter will be set at 0 when the slider-bar is at the 0% value and twice the base value when the slider-bar is at the 100% value. A value

of 0 in this line would indicate that the parameter would be decreased as the value of the personality trait is increased, thereby taking a value of twice the base value when the slider-bar is at the 0% value and 0 when the slider-bar is at the 100% value. Similarly, the `key:`, `base:`, `inc:` lines indicate that the second of the two behavioral parameters controlled by this slider-bar has a key value of `navigation_avoid_obstacle_gain`, a base value of 1.5, and should be decreased as the personality trait is increased. The next group of lines gives similar information for the wanderlust personality trait.

3.1.15 Robot Trails

There are a few miscellaneous options you can toggle from the `Options` menu (Figure 30). The first one, “Show robot Trails”, can make robots leave a black trail on the screen as they move. It is useful when you want to trace the points the robots go through. A shortcut key `Ctrl-t` or specifying “set show-trails on” in a CMDL file will turn on/off this option.

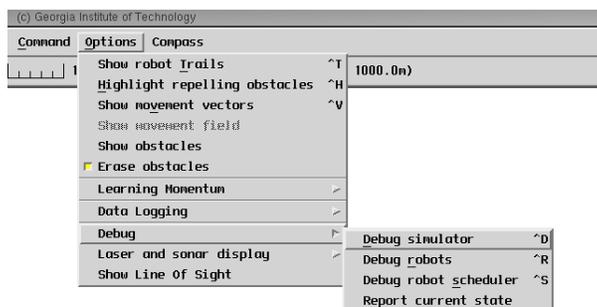


Figure 30: The options and debug menu

3.1.16 Obstacle Highlighting

The second entry, “Highlight repelling obstacles”, in the `Options` menu will turn on/off the option for users to tell whether the robots are in the zone of influence and safety zone for each obstacle. The zone of influence is the area where robots react with the obstacles, and a blue circle will show up when the “Highlight repelling obstacles” option is on. If the robots are in the safety zone, inter-robot collisions are assumed to occur, and it will be indicated with a red circle. You can turn on this option with a shortcut key `Ctrl-h` or specifying “set highlight-repelling-obstacles on” in a CMDL file.

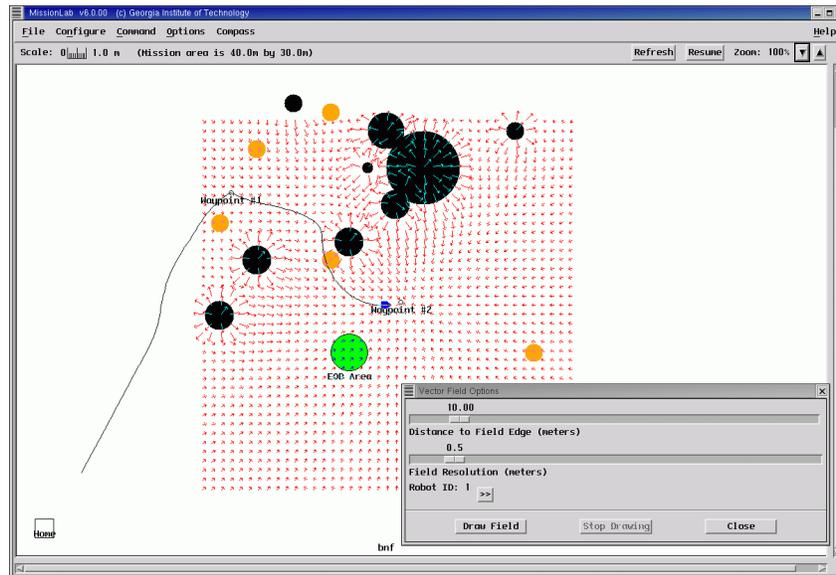
3.1.17 Movement Vector

You can observe a heading direction of each robot by turning on the option “Show movement vector” in `Options` menu. The heading direction will be displayed as a red arrow on each robot in the screen. The shortcut key is `Ctrl-v` and the CMDL file can take “set show-movement-vectors on” command also.

3.1.18 Movement Field

The “Show movement field” option in the `Options` menu is a useful tool that shows the potential field for a robot. Selecting this option will open the “Show movement field” window (Figure 31).

The *Distance to Field Edge* slider lets you dictate the size of the vector field, which is a square centered on the robot for which the field is being drawn. The *Distance to Field Edge* is the distance from the robot

Figure 31: Movement field shown in *mlab*

to the edge of the field in meters, so one side of the vector field is actually twice this length. The *Field Resolution* slider dictates the resolution of the grid on which the field is drawn. To set which robot for which the field should be drawn, use the *Robot ID* button to scroll through available robot IDs.

This function should only be invoked when the simulation is paused. All vectors in the field are based on a snapshot of sensor readings of the robot at its current position (whether running in simulation or on a real robot). Also, since the vectors in the field are generated by running the controller with the same sensor readings over and over again, it is not recommended that this function be used with behaviors that are time-dependent.

3.1.19 Sensor Readings Display

If you are running a mission on a real robot, you might want to turn on “Show obstacles” in the **Options** menu, which allows *mlab* to display objects that are perceived by the robot sensors. If the *mlab* screen is filled with the objects, and if you wish to erase them, you can click on the **Refresh** button on the scale bar manually whenever you want to erase them, or turn on the “Erase obstacles” option in the **Options** menu to automatically erase them after a specified period.

You can also change the shape of the sensor readings by selecting options from “Laser and sonar display” (Figure 32). The followings are the alternative shapes supported by the option:

- “Laser Display Normal”: The laser readings are transformed into the world coordinate and will be displayed around the robot display.
- “Laser Display Linear”: The raw laser readings will be displayed as a graph (X-axis = laser number from 0 to 361, Y-axis = distance from perceived objects) at the lower left corner of the *mlab* screen.
- “Laser Show connected”: The laser readings will be displayed as a line instead of individual points.
- “Show sonar as arc”: The sonar readings will be display as an arc instead of a point.
- “Show sonar as point”: The sonar readings will be display as a point.

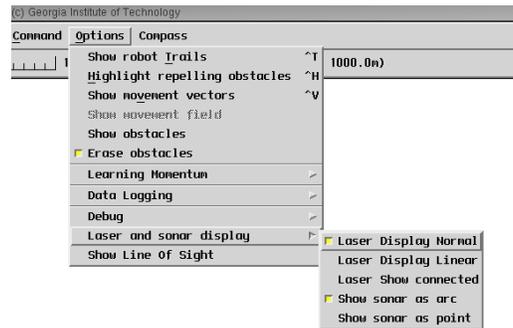


Figure 32: The options for the obstacle display

3.1.20 Data Logging

Upon the real-time execution of the robot, for both simulated and physical robots, you can record the position, velocity, heading, and the current state of the robot with respect to time if the data logging function is invoked. This function can be turned on by both from this “Data Logging” menu and from the *mlab* command option “-L”.

An output file `<robot executable name>.log`. will be then generated by *mlab*. An example of the output file is shown below.

MissionLab Robot Executable Status Data

Time	Position-X	Position-Y	Heading	Velocity-x	Velocity-y	RobotID	State
0.000	39.400	17.300	0.000	0.000	0.000	1	0
0.345	39.400	17.300	0.000	0.000	0.000	1	1
0.513	39.400	17.300	0.000	0.000	0.000	1	2
0.517	39.400	17.300	0.000	-0.018	0.199	1	2
0.522	39.382	17.499	95.059	0.195	0.044	1	2
0.527	39.321	17.690	107.766	0.189	0.065	1	2
0.532	39.201	17.850	126.840	0.181	0.086	1	2
0.537	39.024	17.943	152.299	0.199	0.021	1	2
0.543	38.839	18.017	158.219	0.197	0.035	1	2
0.569	38.643	18.057	168.417	0.197	0.035	1	3

3.1.21 Debug Options

Turning on the debug option in **Options** menu will dump a huge amount of information of what *MissionLab* is doing. In most of the cases, it will print out all the `printf` statements specified as “`if (debug)`” in the *MissionLab* source code to the console where you launched *mlab*. For example, turning on “Debug simulator” (`Ctrl-d` or “set debug-simulator on” in the CMDL file) will provide the information about *mlab* while “Debug robots” (`Ctrl-r` or “set debug-robots on” in the CMDL file) will provide the information about the robot executables. If “Report current state” is turned on, *mlab* will pop-up the window (Figure 33) showing the state the robot is currently in. This “Report current state” window can be also invoked by adding “-S” to the *mlab* command option.

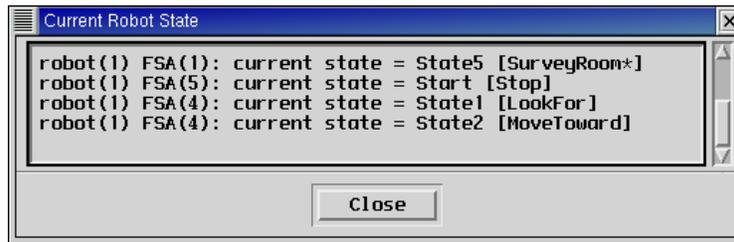


Figure 33: Report Current State Window

3.1.22 Show Line Of Sight

Turning on the “Show Line Of Sight” option enables the graphic visualization of “Visibility chains” composed of robots belonging to a co-operating team. First, we remind the notion of “Visibility”: a robot A can see a robot B (and vice versa) if the line segment that joins A and B lies completely in the free space. Next, we introduce a transitive property in the definition of “Visibility”: if robot A can see robot B and robot B can see a third robot C, we say that A can see robot C. The notion of “Visibility” is very important in many multi-robot applications, whenever we ask robot to accomplish missions in an unknown, potentially hostile environment where standard radio communication is not possible and only line-of-sight communication is allowed [18]. A “Visibility chain” is defined as a group of robots that can see each other (and, consequently, communicate with each other) according to the last definition of visibility that has been given (the one with the transitive property). Since the robots deployed in an unknown environment are often given the task of exploring it and reporting to a base station what they have discovered, maintaining “Visibility chains” with the base station is very useful to quickly communicate what they have found during exploration. In order to visualize “Visibility chains” the user must create a robot whose color is “yellow”; assigning the “yellow” color identifies the robot as the base station (or a robot in a safe area to which all information must be reported). All robots belonging to a “Visibility chain” together with the “yellow” robot are turned into “yellow” robots. They are turned back to their original color if the “Visibility chain” with the “yellow” robot is broken. Currently, this option can be used only in simulation.

3.1.23 Compass Window

Figure 34 is the compass window which can be invoked by “Compass” menu in the menu bar. When a robot is running, an arrow will show up in the circle area indicating the heading direction of the robot with respect to the compass direction. This function works for both the simulation and the robot that supports a compass device. Press **End Compass** to close the window.

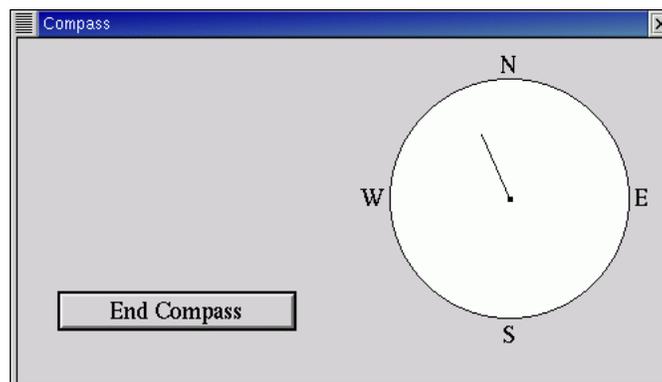


Figure 34: Compass Window

3.1.24 3D Display

MissionLab has the capability to represent three dimensional heightmapped terrain. In order to utilize this feature, a heightmap datafile must exist in the current working directory. This file should be an 8-bit raw greyscale image. The standard linux utility ImageMagik is capable of converting images to this format (e.g., run “convert filename.tif filename.gray”, and then “mv filename.gray filename.hm”). In this image, light areas correspond to elevated regions and dark areas correspond to lower elevation. In order to use this heightmap, a heightmap command should be included in the overlay file. The format of the heightmap command is as follows:

```
HEIGHTMAP heightmap_filename heightmap_width heightmap_height sea_level scale
```

The heightmap width and height should match that of the mission area. The sea level parameter indicates that all elevation units equal to or below that value are water. The scale parameter should, under normal circumstances, be 1.0.

The three dimensional view of the mission area can be accessed during mission execution through the Show 3d Visualization dialog under the Options menu. The 3d scene can be translated, scaled, or rotated by dragging the mouse while clicking the left, middle, and right button respectively.

A sample mission in which the robot navigates in a 3D environment (Figure 35) can be found in [*your MissionLab home*]/demos/ao-fnc.demos/. Run a script called `demo_3d`, compile it, and run the mission after enabling the `Enable 3D` option in the run dialog.

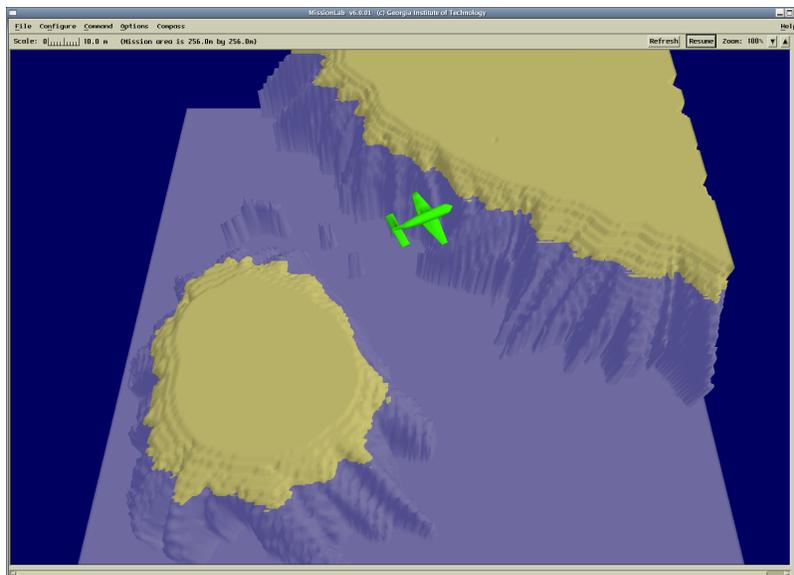


Figure 35: *mlab* - the 3D Display

3.1.25 Help Menu

The help menu, located on the right end of the menu bar, has two entries: About and Copyright that each invoke explanatory dialog boxes. The “About” dialog box gives information about the version of the program and the software creators. The “Copyright” dialog shows the full copyright agreement protecting this software.

3.2 Robot Executables

When you run *MissionLab* whether for simulation or real robots, *mlab* (Section 3.1) forks a program called “robot executable” for each robot. The robot executable contains two main libraries that are used to carry out a mission. The first library (*libcnl.a*) stores a set of the primitive behaviors, coded in the Configuration Network Language (CNL). These primitive behaviors are based on Motor Schema theory [2] and details are explained in [1]. The other library (*libhardware_drivers.a*) stores a set of functions that communicate with the *mlab* console and, if it is for the real robots, give commands to the robot hardware or to the low-level software embedded on the robot using Hardware Server or *HServer* (Section 3.8) . In this case, the robot executable serves as *HClient*, and talks to *HServer* using IPT (Section 4.1).

There are currently two methods to create the robot executable. The first method is through the Configuration Editor (*CfgEdit*). *CfgEdit* can create the robot executable with a graphical user interface. Thus, you do not have to do any coding. *PatrolCarRobot1* and *PedestrianRobot1* in the [*your MissionLab home*]/demos/tmr_demos are examples of this *CfgEdit*-type robot. Details on *CfgEdit* are explained in Section 3.3. The other method to create the robot executable is by programming in CNL manually. *mlab-2.0-robot* ([*your MissionLab home*]/src/robot/mlab-2.0-robot/mlab-2.0-robot) and *tmr_robot-1998* ([*your MissionLab home*]/src/robot/tmr_robot-1998/tmr_robot-1998) are examples of this manually-coded-type robot. The main difference between those two types is that the *CfgEdit*-type contains the entire mission of the robot in its program. In other words, a mission from the start-state to the final stop-state (or terminate-state) and their triggers that invoke the state transitions are all encoded in the executable file before it is executed. On the other hand, the manually-coded-type robot does not contain information about the entire mission. The state transitions of robots rely on commands described in a command description file (CMDL file) of *mlab*, or inputs from the *mlab* Command Interface. The manually-coded-type robot is implemented with a set of behaviors for moving to a specified location, and occupying a position. A group of these robots can use several formation behaviors, such as line formation, column formation, diamond formation, wedge formation, and no formation.

3.3 CfgEdit (Configuration Editor)

As you have observed with the demo program in Section 2.2.2, *Cfgedit*, or the Configuration Editor, is a graphical tool for building a mission with a set of robot behaviors. It can then translate the mission as source code and compile it to create a robot executable that can directly control a simulated or real robot.

3.3.1 Reviewing the Configuration File

Before running *CfgEdit*, it would be useful to review the configuration file that specifies the configuration of *CfgEdit*. You can find the default configuration file (`.cfgeditrc`) in `[your MissionLab home]/src/cfgedit2`. Note that, since *MissionLab* 7.0, customization of the paths in this file is no longer necessary as the path for the individual *MissionLab* home is automatically acquired through the `configure` script during the installation (see `$MLAB_PATH` in `src/make.include`).

The following is the sample of the configuration file for *CfgEdit*, which is included in the *MissionLab* package:

```
# Make backup CDL files from the editor (true or false).
backup_files = false

# Show the values of the slider bars instead of the symbolic names
ShowSliderValues = True

# Hide the parameters for a state/trigger of FSA in detail for default. (true or false)
HideStateTriggerParameters = True

# Disable the Waypoints function in the main menu (Note: Not MissionExpert). (true or false)
DisableWaypointsFunction = false

# Disable Path Planner. (true or false)
DisablePathPlanner = true

# Disable Honeywell Real-Time Analyzer. (true or false)
DisableHoneywellRTAnalyzer = true

# Disable MissionExpert. (true or false)
DisableMissionExpert = false

# Disable MissionExpert's repairment feature. (true or false)
DisableMissionExpertRepair = false

# Hide the MissionExpert menu even if the function itself enabled. (true or false)
HideMissionExpertMenu = false

# Disable AssistantDialog. (true or false)
DisableAssistantDialog = true

# Hide AssistantDialog upon starting of CfgEdit. (true or false)
HideAssistantDialog = false

# Specify where the message file for Assistant Dialog is stored.
AssistantDialogMsgDir = $MLAB_HOME/src/assistantDialog/MessageFiles/

# Specify where 3D models are saved.
```

²The symbolic link is pointed to this file from `[your MissionLab home]/bin`.

```
Vis3DModelDir = $MLAB_HOME/src/terrain/models

# Set the capabilities of the user.
# Need Execute in order to run a configuration
# Need Modify in order to be able to modify parameters in a configuration
# Need Edit in order to be able to create or modify configurations
# Need Create in order to be able to create new components
# Need Library in order to be able to modify library components
# Need RealRobots in order to be able to run real robots
# Need MExpModifyCBRLib in order to be able to modify the library of the CBRServer.
#
user_privileges = Execute, Modify, Edit, Create, RealRobots, Library, MExpModifyCBRLib
#user_privileges = Execute, Modify, Edit, Create, RealRobots, MExpModifyCBRLib
#user_privileges = Execute, Modify, Edit, Create, RealRobots, Library

# Select which primitives are shown to the user
# List architectures here to restrict names to only those names occurring
# in all of the listed architectures. The default is to show all names.
# Architectures:
#   AuRA
#   AuRA.urban
#   AuRA.naval
#   UGV
#
restrict_names = AuRA.naval

# List of comma separated directories and root names of the
# CDL description libraries to load.
# Will try to load xxx.gen, xxx.AuRA, and xxx.UGV
CDL_libraries = $MLAB_HOME/lib/default,
               $MLAB_HOME/lib/agents

# Optional: Configuration to load as the empty config.
#DefaultConfiguration = $MLAB_HOME/lib/FSA.cdl
DefaultConfiguration = $MLAB_HOME/lib/default.cdl

# Where to find the map overlays
MapOverlays = .

# Default Overlay
DefaultOverlay = Empty.ovl

# Default remote host
#DefaultHost = cortez.cc.gatech.edu

# Directory of the remote machine where the robot executable should be
# FTP-ed to, and executed from. (Note: It only affects if you specified
# a machine name in the "Remote host computer" field of the run dialog.)
RemoteHostRobotDir = /home/demo

# The user name for executing the robot executable on a remote machine.
# If it is commented, the name of the user who are running mlab will
# be used. (Note: It only affects if you specified a machine name in
# the "Remote host computer" field of the run dialog.)
#RemoteShellUserName = demo

# The machine name that laserfit is running on.
```

```
LaserFitMachineName = columbus

# Default robot color
DefaultColor = blue

# Default HServer ID that the robot executable and HServer connect each other via IPT
DefaultHServerName = fred

# Directory to write the event logs.
# None are written if this is empty.
#EventLogDir = .

# Directory to write the event replay CDL files. The event logging (above) has to be on.
# None are written if this is empty or the event logging is off.
#ReplayDir = .

# Directory for saving the Mission Expert output data files.
# The files will be deleted if this option is commented.
#MExpDataFilesDir = .

# Name (with path) for the socket of CBR Planner.
#CBRServerSocketName = /tmp/robot-cbrplanner.socket

# This option specifies whether to disable premission-CNP upon specifying
# a mission using Mission Expert. Also, this option should be set to False
# (i.e., enabling Premission CNP) if you want to display robots based on
# the configuration specified in ROBOT-CONSTRAINTS section of .mission_expert.rc.
DisablePremissionCNP = False

# This option specifies whether to disable runtime-CNP upon specifying
# a mission using Mission Expert.
DisableRuntimeCNP = False

# This option specifies whether to allow the user to alter the robot constraints
# via the Mission Expert's user interface. If this set to True (as it should be),
# the robot constraints specified in .mission_expert.rc by "experts" will be used.
DisallowModCNPRobots = True

# The first robot's start position. It has to be defined as PP. If it is commented,
# default "StartPlace" will be used.
FirstRobotStartPlace = StartPlace

# Offsets to start robots at. This is for the 2nd and later robots in the multi-robot mission.
# The first robot won't be affected. Not for the extra robots.
RobotStart_dx = 1
RobotStart_dy = 0

# Extra robots to add to the run.
#ExtraRobots = "Enemy \\.\/robots\/SentryRobot1\" "red" (robot_type = HOLONOMIC, run_type= SIMULATION, extra_robot = true)
#ExtraRobots = "Enemy \\.\/enemyRobot1\" "red" (robot_type = HOLONOMIC, run_type= SIMULATION, extra_robot = true)"

# The extra robot's start position. It has to be defined as PP. If it is commented,
# it will be treated as one of the multi-robot (i.e., having RobotStart_dx RobotStart_dy
# offsets).
#ExtraRobotStartPlace = SentryPost
ExtraRobotStartPlace = EnemyStartPlace

# Offsets to start extra robots at. This is for the 2nd and later extra robots.
```

```
# The first extra robot won't be affected. If these are uncommented, RobotStart_dx
# and RobotStart_dy above will be used.
ExtraRobotStart_dx = 1
ExtraRobotStart_dy = 0

# Whether to enable the robot data logging. (Not the usability or event log.)
LogRobotData = False

# Whether to show the current status of the robot when mlab is running
ShowRobotStatus = False

# To set the default time (second) for the mission to expire, uncomment this line.
#MissionExpirationTime = 180

# ***** CNL Architecture configuration *****

# Directories to look in for link libraries.
# Seperate each path with a colon. Newlines are allowed after the colon.
lib_paths = $MLAB_HOME/lib

# List of comma separated directories and root names of the
# CNL source files and libraries to load.
# Will try to load libxxx.a and include xxx.inc as the cnl header file
# The editor will look for yyy.cnl in these locations for each extern agent
CNL_libraries = $MLAB_HOME/lib/cnl

# directories to look in for CNL source files to display in the editor.
CNL_sources = $MLAB_HOME/src/libcnnl

# cflags parm passed to C++ compiler
cflags = -g,
        -I$MLAB_HOME/src/include

# ldflags parm passed to C++ compiler
ldflags = -L$MLAB_HOME/lib,
        -lcthreads,
        CNL_LIBS,
        -lhardware_drivers,
        -lipt,
        -lstdc++,
        -lm,
        -lqlearn,
        -lcbr_behavioral_select,
        -ldstar,
        -lcnp,
        -lCommBehavior

# ***** Real robot configuration flags *****
#The list of real robots we know about
robots = HServer

# Any misc robot settings that will be dumped to the script file
MiscRobotSettings = "set show-trails on",
        "set scale-robots on",
        "set ROBOT-LENGTH 1"

# Any list of strings attached to the robot name will be appended
# to the startup parameters for that robot. The robot names are
```

```
# case sensitive.

HServer = "robot_type = HSERVER",
"danger_range = 1.0",
"compass_type = NOCOMPASS",
"use_reverse = 0",
"draw_obstacles = 1",
"lurch_mode = 0",
"wait_for_turn = 0",
"drive_wait_angle = 90",
"adjust_obstacles = 1",
"use_sonar = 1",
"use_laser = 1",
"use_cognachrome = 1",
"wait_for_ack = 0",
"multiple_hservers = 0"
```

3.3.2 Running *CfgEdit*

CfgEdit can also take several optional arguments. Before you run *CfgEdit*, make sure you are running *iptserver* (Section 3.1) and “Empty.ovl” (or another default overlay file you specified in the line “Default-Overlay” of your configuration file) is in the directory from where you will run the *CfgEdit* from, so that when you run *mlab* within *CfgEdit*, it will run properly. You can find “Empty.ovl” in [*your MissionLab home*]/demos/mars_demos and/or [*your MissionLab home*]/src/overlays directory. The full form is:

```
cfgedit [options] [CDL file]
```

As for *mlab*, all of the arguments are optional. The main argument, *CDL file*, is the name of a file written in the Configuration Description Language (CDL). *CfgEdit* creates a CDL file when you save a mission with “Save Configuration As” from the file menu. The CDL file have an extension of “.cdl”. The following options are supported by *CfgEdit*:

- a Enable the automatic execution. If used with the *Mission Expert* mode, the whole mission specification process including compilation and execution can be automated. Useful for an experiment using the CBR Wizard.
- c *filename* Use a specified configuration file instead of .cfgeditrc.
- e *filename* Enable event logging. Use the specified file.
- l Turn on the debugging mode for the CDL parser. This option will print out debugging information for the CDL parser (yyparse). *CfgEdit* uses this parser to process the CDL code in order to extract the information needed to generate CNL (Configuration Network Language) code. This option and “-s” option below are probably not very helpful to anyone but the developers.
- M *string* Enable the Mission Expert mode. The specified ‘string’ is to name output files.
- s Enable debugging of the *cdl* code generator. When you try to “Compile” a robot executable with *CfgEdit*, at first the CDL code will be translated into CNL code by the *cdl* code generator. This option prints out the debugging information during the translation process.
- v The verbose mode. This option will print out the modes you set up in the “.cfgeditrc” file when you run *CfgEdit*.

3.3.3 Browsing the Configuration Tree

After invoking *CfgEdit* with the command above, you will see the main window and a pop-up window asking you to choose one of the three options: “New Robot”, “Load Robot”, and “Quit” (Figure 3 on Page 7). If you choose “New Robot”, *CfgEdit* will load the configuration specified in “.cfgeditrc” (unless your version of *MissionLab* was installed with the Case-Based Reasoning Mission Planning Wizard capability). If you choose “Load Robot”, *CfgEdit* will ask you to select a previously saved configuration (CDL file). You can also load a saved configuration from by selecting “Open” from the “File” menu from the menu bar on top of the window. Choosing “Quit” will exit *CfgEdit*.

The first box you see in the window when you start *CfgEdit* (Figure 4 on Page 8) is the top level of the configuration tree. As you can see from the diagram in Figure 36, this level combines numbers of individual robot configurations as a group. In other words, *CfgEdit* is implemented with the capability of creation, configuration, compilation, and execution of missions for multiple robots. For example, if you wish to create two robot executables which has the same mission and configuration, go to the 2nd level by middle-clicking the “Kind of: Assemblage” box, highlight the “Kind of: Robot” box by left-clicking it, copy it by selecting “Copy” from the “Edit” menu in the menu bar, and paste it by selecting “Paste” from same “Edit” menu. Note that the box that has just been pasted will overlap with the original one, so you may want to move it to somewhere in the screen by dragging it (Figure 37). You can also replace the procedures above with selecting “Duplicate” from the “Edit” menu. If you wish to go back to see one level higher than the level where you are, you can left-click the Up 1 Level button in the left menu bar.

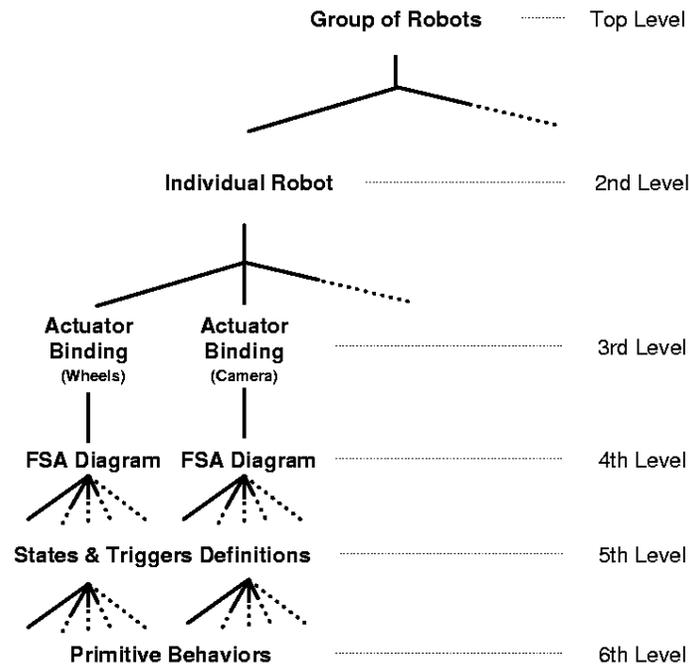


Figure 36: *CfgEdit* - a typical configuration tree.

The third level of the configuration tree describes what state machine is binding to each actuator. This level can be reached by middle-clicking the “Kind of: Robot” box from the second level. The default configuration (default.cdl) is bound with two actuators, wheels and camera (Figure 5).

If you middle-click either of the “Instance of: FSA” boxes, you can reach the Finite State Acceptor (FSA) diagram (the fourth level). Each of the actuators has its own FSA diagram and you can construct a mission using a set of pre-defined tasks/states and triggers (Figure 6). The construction of a mission using this FSA diagram is explained in Section 3.3.4.

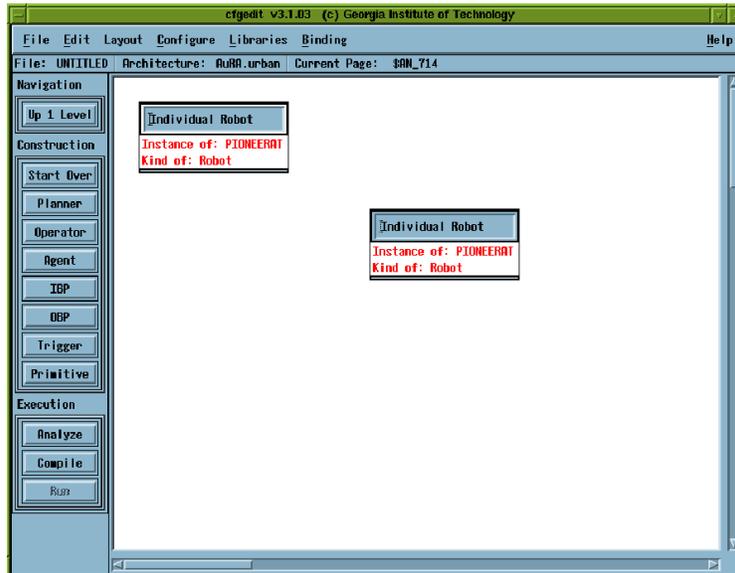


Figure 37: *CfgEdit* - the second level with two robot-configurations.

The information specified up to this point (from the top level to the fourth level) is stored in a CDL file. This is in plain text format so you can view this file with any editor. The details of CDL are explained in Section 3.4.

To find out how each state and trigger is defined, you may want to go to the fifth level by shift-middle-click (while pressing the `Shift` key, click the item with the middle button of the mouse). Middle-click the next box which has a label “Move to the specified location”, and now you should see a figure of four boxes (Figure 38). This figure shows that the “GoTo” state generates the heading by taking the summation of the vectors towards the goal location, away from the obstacles, and towards the direction the user specified using Telop. By middle-clicking any of the three “Kind of: Assemblage” boxes (for example, the one for “Move the robot to the goal” in this case), you can find that this vector was computed by the primitive behavior “MOVE.TO” or the “Move-to-goal” schema [1] after the goal location was defined (Figure 39). The definition of the other states and triggers that can be obtained by shift-middle-clicking them at the FSA diagram (the fourth level). One thing you may want to note is that, unlike the information you specified in the FSA diagram or above levels, the definition of each state and trigger cannot be, generally, modified with the *CfgEdit* window unless you import it to your system. These definitions are defined in `[your MissionLab home]/src/cdl_code/agents.AuRA.naval` (also linked from `[your MissionLab home]/lib`), not the CDL file that ends with “.cdl”. When you select to `Compile` the robot executable, at first by default this “agents.AuRA.naval” file will be bound to the CDL file.

Some states, however, contain nested sub-FSA states and triggers. In other words, a state is composed with a set of other states and triggers. Figure 40 shows, for example, nested sub-FSA states and triggers for the “SurveyRoom*” state. You can observe them by shift-middle-click the state once, and middle-click the next two boxes. Even though, the definition of the state with nested sub-FSA states and triggers are defined in `[your MissionLab home]/src/cdl_code/agents.AuRA.naval`, you can rearrange those after importing them to your system.

Finally, *CfgEdit* is also capable of showing how each primitive behavior is coded. The primitive behaviors are written in CNL, and stored in `[your MissionLab home]/src/libcnl` directory as a library (`libcnl.a`). Upon compilation, it will be linked to the robot executable. For example, to see the CNL code for the “MOVE.TO” behavior, middle-click the “Instance of: MOVE.TO” box (Figure 39), and the CNL code will pop up as shown in Figure 41. See Section 3.5 for details on CNL.

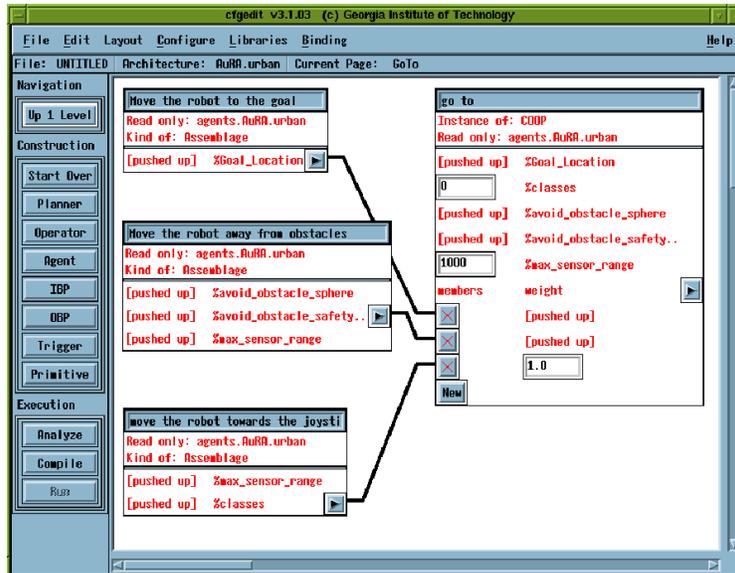


Figure 38: *CfgEdit* - the definition of the “GoTo” state.

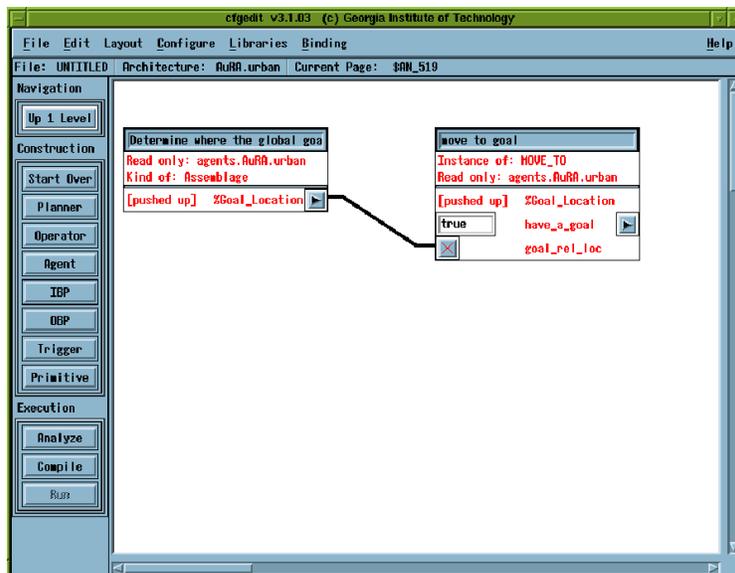


Figure 39: *CfgEdit* - the definition of the “Move the robot to the goal”.

3.3.4 Building a Mission

Construction of a mission using the FSA diagram can be done at the fourth level of the configuration (Figure 6 on page 9). When you first reach this level, you find a “Start” state. To add a new state, left-click the **Add Tasks** button located in the left menu bar, move the pointer to the display area, and left-click again. The default new state is a “Stop” state (Figure 7 on page 10). Right-clicking the state icon will popup a list of available states. By default, the system is binded to AuRA.naval architecture, and available states are limited to naval applications. To change the binding architecture to, for example, AuRA.urban, and create a mission using behaviors for urban applications, you can modify the configuration file (.cfgedit) and set `restrict_names` to be `AuRA.urban` and `DefaultConfiguration` to be `urban.cd1`. The full list of the available states in *MissionLab* is shown in Table 1.

If you create more than one state in the mission, you can connect them with triggers. Once the condition specified in the trigger is satisfied, the robot will transition to the next state to which the trigger is pointed (Figure 11 in Page 12). To connect two states, left-click the **Add Triggers** button located on the left menu bar, press down the left button of the mouse on the state where you want to have the transition from when the condition is satisfied, drag the arrow to the next state, and release it. The modification of triggers is done in the same manner as for the states above. The full list of the available states in *MissionLab* is shown in Table 2.

For building the mission, all the edit functions “Copy”, “Duplicate”, “Cut”, “Paste”, and “Delete” in the “Edit” menu are supported by *CfgEdit*. If you make a mistake during the construction of the robot mission and you wish to start over, left-click the **Start Over** button located on the left menu bar. *CfgEdit* will, then, clear the entire mission, and start a new configuration without killing its process.

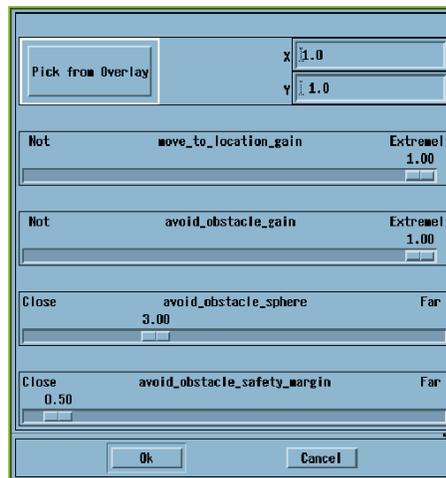


Figure 42: *CfgEdit* - modifying parameters for the GoTo state

3.3.5 Using Waypoints Tool

If you want to create a robot mission in which the robot navigates the environment via a sequence of points, you might find this “Waypoints” tool useful. This function can be invoked by the **Waypoints** button located on the left side of the screen during the construction of a FSA mission.

This waypoint function converts the *mlab* console into the waypoints-creation mode. After choosing an appropriate overlay from the menu, waypoints can be selected by just clicking the left mouse button on the window (Figure 43). Middle clicking will allow you to delete an added waypoint when it was created

Table 1: Available States in *MissionLab*

STATE	DESCRIPTION	Supported by
AboutFace	The robot faces in the opposite direction from the original.	AuRA.naval AuRA.urban
Alert	The robot alerts the user by sending a specified message. The user will received the message by the pop-up window on the <i>mlab</i> console, and, if specified, by email.	AuRA.naval AuRA.urban
AssistedGoTo	The robot behaves as same as the GoTo state (explained below) as long as the robot is progressing towards the goal. However, if the robot is making no progress, the robot behaves as same as the Telop state (explained below), so that the user can assist the robot moving towards the goal using the Teleautonomous Operation (<i>Telop</i>). This state is composed with a set of sub-FSA states and triggers: Start , Immediate , GoTo , NoProgress , Telop , and TelopComplete .	AuRA.urban
CNP_BidOnTask	The robot bids on a CNP task.	AuRA.naval
CNP_ExecuteWonTask	The robot execute the CNP task that it just won.	AuRA.naval
CNP_InjectTask	The robot injects a CNP task to the CNP initiator.	AuRA.naval
CNP_NotifyTaskCompleted	The robot notifies to the console that the CNP task was completed.	AuRA.naval
CNP_NotifyTaskReneged	The robot notifies to the console that the CNP task was reneged.	AuRA.naval
CNP_NotifyWonTaskStarted	The robot notifies to the console that the CNP task that just won was started.	AuRA.naval
CNP_SaveDynConstraints_Track	The robot saves the dynamic constraints for CNP Track task.	AuRA.naval
CNP_SaveTargetLocation	The robot saves the location of the target for CNP.	AuRA.naval
CNP_SaveTargetVehicleType	The robot saves the vehicle type of the target for CNP.	AuRA.naval
CNP_SaveTargetVelocity	The robot saves the velocity of the target for CNP.	AuRA.naval
ChangeMotVector	This task will increase/decreases the values for the motivational vector variables. See Section 3.1.12.	AuRA.urban
ChangeRobotColor	This task will change the display color of the robot in <i>mlab</i> .	AuRA.naval AuRA.urban
DropObject	The robot puts the carrying object into the closest object.	AuRA.naval
EnterAleternateHallway	The robot enters an adjoining hallway using the docking behavior [1]. The direction of the robot heading is chosen randomly after passage.	AuRA.urban
EnterRoom	The robot enters a room through the nearest door using the docking behavior [1]. It can be specified to enter the room through either an unmarked door or any door.	AuRA.urban
ExitTask	This state is usually used in a sub-FSA state. For example, the SurveyRoom* state (Figure 40) contains two ExitTasks to reflect the two conditions for the robot to finish the surveying room task: one for when the robot did not find a biohazard in the room, and the other one for when the biohazard was detected. ExitTask takes a string to notify the main-FSA. When the string of the TaskExited trigger in the main-FSA matches with the ExitTask 's string, the robot can move on to next state which the TaskExited trigger is pointing to.	AuRA.naval AuRA.urban
Follow	The robot follows either a friendly robot or an enemy robot.	AuRA.naval AuRA.urban

STATE	DESCRIPTION	Supported by
FollowCMDLiCommands	The robot follows the CMDLi commands.	AuRA.urban
FollowCSBAdvise	The robot follows the communication sensitive behavior advise.	AuRA.urban
GoThroughDoor	The robot enters a nearest door it finds.	AuRA.urban
GoTo	The robot moves to a location specified in X - Y coordinates. You can type in the location as well as pick them from an overlay. If you left-click the <code>Pick from Overlay</code> button, <i>mlab</i> will pop up and ask you which overlay you want to use. Once the overlay is selected, you can left-click a desired position and it will be automatically translated into the X-Y coordinate. As you can see from Figure 42, some states (e.g., GoTo and MoveToward below) have the option that you can modify the gains of their primitive behaviors that are incorporated into the state. For example, if you increase “move_to_location_gain”, the attraction to the goal becomes larger while increasing “avoid_obstacle_gain” will cause repulsion from obstacles to be larger. The other parameters you can modify from this window are sizes for the sphere of influence and safety sphere for the obstacle avoidance behavior: “avoid_obstacle_sphere”, and “avoid_obstacle_safety_margin”, respectively. The zone of influence is the area where robots react with the obstacles, and if the robots are in the safety zone, virtual collisions are assumed to occur. To see how gains are set in your constructed FSA, toggle “Show state/trigger parameters” function in the third entry of the “Layout” menu.	AuRA.naval AuRA.urban
GoToAvoidPast	The robot moves to the specified location while being repulsed by locations which it has already visited [5]. This allows the robot to explore the environment by escaping from local minima. In particular, the robot stores in memory a map of the environment (a two-dimensional grid) where visited and unvisited locations are assigned different values; as the robot visits an area more times, the value of the corresponding cells in the grid increases and, consequently, the repulsive force exerted by the cells increases as well. In addition to the parameters of the GoTo state, the parameters you can modify are the gain for the “avoid the past” behavior (avoid_past_gain), the number of cells around the robot that are considered when computing the repulsive force (avoid_past_horizon correspond to the side of a square area centered on the robot), the number of cells around the robot that are marked as visited locations at each computation step (avoid_past_mark correspond to the side of a square area centered on the robot), the dimension of the grid that the robot stores in its memory (avoid_past_grid_size correspond to the side of a square area). Consider that, whenever the robot exits the grid, a new grid (centered on the robot) is stored in memory which only partially overlaps with the previous one. Consequently, all the information stored in the non-overlapping area gets lost. The scale of the grid is fixed and set to 10 cells/meters.	AuRA.urban

STATE	DESCRIPTION	Supported by
GoToOutdoorNavigation	The robot moves to the specified location with a swirling motion while avoiding obstacles.	AuRA.naval AuRA.urban
GoToOutdoor_CBR	This behavior is based on the GoToOutdoorNavigation behavior. As explained for GoTo_CBR (below), the behavioral parameters are determined by a Case-Based Reasoning system (Note: this CBR is totally independent from the CBR used in <i>CBRServer</i>). The only difference is that the avoid-obstacle behavior is substituted by the swirl behavior.	AuRA.urban
GoToSoundSource	The robot moves in the direction of the sound source.	AuRA.urban
GoTo_CBR	This behavior is based on the normal GoTo behavior. Instead of the user handcoding the behavioral parameters, it uses a Case-Based Reasoning system for Behavioral Selection [11][12] (Note: this CBR is totally independent from the CBR used in <i>CBRServer</i>). The system keeps a set of such parameters in a library, and chooses the one best fitting the current environment. The library can be modifiable or hand-coded. In the first case, the system can learn starting from an empty library, or improve an existing library. Please note that the case library file needs to be in the same directory as the robot executable. An empty library is provided at <i>[your MissionLab home]/-demos/mars_demos/cbr_behavioral_select.cases</i> .	AuRA.urban
GoTo_Dstar	This behavior implements the D* Lite planner within the GoTo behavior. For more details on D* Lite, please refer to the paper by Koenig and Likhachev [8]. The behavior is a front-end for an architecture that tries to switch between planning (by D* Lite) and reactive control at appropriate situations. The two parameters that control the switching capability are the “Dstar_angle_dev” and “Dstar_persistence”. The “Dstar_angle_dev” parameter is a measure of the ‘tolerance’ of the planner or the freedom given to the reactive control by the planner. A small value for this parameter (which is an angle with unit in degrees) causes the planner to remain in control most of the time while a large value leaves the reactive control in charge. The “Dstar_persistence” parameter determines the inertia of the planner. The planner is activated only N cycles after the condition for its activation becomes true (and then only when the condition is still true after the N cycles), where N is the value of the “Dstar_persistence” parameter. The behavior of the system when the “Dstar_persistence” is set to zero is undefined. Upon successful completion of a mission, the behavior outputs a map in the file named “dstar.map”, in the same directory as the robot executable. This map is a text file where each character represents a grid cell. During any run, the behavior first determines if this file exists and loads it as an initial map if so. This file has to be deleted if another mission is to be run on a different overlay. Future releases of <i>Missionlab</i> will include the capability of creating a map for D* Lite directly from an overlay. A feature that determines if the map in the directory corresponds to the current environment will also be included.	AuRA.urban

STATE	DESCRIPTION	Supported by
GoTo_LM	This behavior is an alteration of the GoTo state. Instead of using static values for the state parameters, the move-to-goal gain, avoid-obstacle gain, wander gain, avoid-obstacle sphere-of-influence, and wander persistence are continuously changed at mission run time using a Learning Momentum (LM) algorithm [3], [10]. The LM strategy implemented by default is <i>ballooning</i> . Currently, only one GoTo_LM state is supported per robot at one time. If more than one GoTo_LM state is used, the behavior is undefined. To make alterations to the LM algorithm (for example, to switch to a <i>squeezing</i> strategy), it is necessary to alter the source file (mlab directory)/src/hardware_drivers/lmadjust.c, do a “make all” in the src or src/hardware_drivers directory to recompile the hardware_drivers library, and recompile the robot in <i>cfgedit</i> . The most relevant part to change in <i>lmadjust.c</i> is the table in the <i>SetDefaultParams()</i> function. The format of this array should be relatively straight-forward having read and understood the LM papers [3], [10]. Future versions of <i>MissionLab</i> will have a new LM integration technique that will remove the single LM state limitation, remove some of the requirements to re-compile parts of <i>MissionLab</i> , and allow for the user to specify the application of LM to many different types of states by altering CDL code (or at a lower level, CNL code) during mission specification.	AuRA.urban
GoTo_MiNav	The robot moves to the specified location while exploring the environment according to the MicroNavigation behaviors [16]. In particular, the robot can be in two states: Avoid Obstacle and Follow Contour. When in the Avoid Obstacle state, the behavior of the robot is very similar to the GoTo state. When in the Follow Contour state, the robot follows the contour of obstacles until it thinks it safe to start heading again towards the goal. The transitions between the two states are determined by two triggers; this allows the robot to explore the environment until it finally reaches the goal. The parameters that you can modify are the sphere of influence of obstacles (<i>avoid_obstacle_sphere</i>), the maximum and minimum level of “bravery” of the robot in approaching obstacles when following their contour (the higher is the value of <i>u_max_val</i> , the closest to obstacles the robot is allowed to go; the lower is the value of <i>u_min_val</i> , the farther from obstacles). The current distance from obstacles is autonomously determined in relation to “ <i>u_max_val</i> ” and “ <i>u_min_val</i> ” according to the MicroNavigation rules, in order to allow a deeper exploration of the environment when the robot cannot find a path to the goal.	AuRA.urban
InitializeCSB	The robot initializes the communication sensitive behavior.	AuRA.urban
Intercept	The robot intercepts a moving object.	AuRA.naval

STATE	DESCRIPTION	Supported by
LOS_GoTo	Currently, all the behaviors with the LOS (Line Of Sight) prefix can be used only in simulation. The behavior is similar to the GoTo state: however, this is meant for multirobot applications, whenever two (or more robots) can be considered as members of a cooperating team. When no teammates are visible, the robot moves towards a location specified in X - Y coordinates. If a robot comes in sight of another robot, they share information about their own targets (the spatial locations they wish to reach) and about their progress towards the goal in order to help each other to accomplish their missions (eventually getting out from local minima) [18]. To implement this mechanism, each robot must be assigned a color which univocally identifies it (see the SetRobotColorId state described in the following). The parameters you can modify are the ones described for the GoTo state; moreover you can specify the robots which should be considered as teammates (by enabling the corresponding colors).	AuRA.urban
LOS_GoTo_AvoidPast	The robot cooperates with other robot whenever they come in sight of each other (as in the LOS_GoTo state); however, when it is not receiving help from teammates, the robot behaves as if it were in the GoTo_AvoidPast state. The parameters you can modify are the ones described for the GoTo_AvoidPast state; moreover you can specify the robots which should be considered as teammates (by enabling the corresponding colors).	AuRA.urban
LOS_GoTo_MiNav	The robot cooperates with other robot whenever they come in sight of each other (as in the LOS_GoTo state); however, when it is not receiving help from teammates, the robot behaves as if it were in the GoTo_MiNav state. The parameters you can modify are the ones described for the GoTo_MiNav state; moreover you can specify the robots which should be considered as teammates (by enabling the corresponding colors).	AuRA.urban
LeaveRoom	The robot leaves the room behind the nearest door using the docking behavior [1].	AuRA.urban
Localize	The robot localizes its current position to be the specified values.	AuRA.naval AuRA.urban
LookFor	The robot moves back and forth until it sees the specified object. This is another example of a state with nested sub-FSA states and triggers. It is composed with: Start , Immediate , MoveForward , MovedDistance , AboutFace , AboutFaceCompleted , Wait , Stops , and Detect .	AuRA.naval AuRA.urban
Mark	The robot marks the nearest object to be the specified object.	AuRA.naval AuRA.urban
MarkDoorway	The robot marks the nearest doorway. This state is useful when the user wants the robot not to go through the door it already went through once.	AuRA.urban
MotivationalUpdate	This task will update the values of the motivational variables of the robot, so that the Motivational Vector interface can update the values of its sliders automatically. See “MOTIVATIONAL_UPDATE.cnl” for how the update rules are set up.	AuRA.urban

STATE	DESCRIPTION	Supported by
MoveAhead	The robot moves to the compass direction specified by the user (East = 0, and North = 90).	AuRA.naval AuRA.urban
MoveAway	The robot moves away from selected types of objects.	AuRA.naval AuRA.urban
MoveForward	The robot moves in the direction it is currently facing.	AuRA.naval AuRA.urban
MoveInFormation	A group of robots moves to a goal location in a specified formation.	AuRA.urban
MoveToward	The robot moves to the closest object of a certain type.	AuRA.naval AuRA.urban
MoveToward_LM	This behavior is an alteration of the MoveToward state. Similar to the GoTo_LM state, instead of using static values for the state parameters, the move-to-goal gain, avoid-obstacle gain, wander gain, avoid-obstacle sphere-of-influence, and wander persistence are continuously changed at mission run time using a Learning Momentum (LM) algorithm. See the description for the GoTo_LM state above.	AuRA.urban
MoveToward_NotifiedObject	The robot moves to the broadcasted object location.	AuRA.naval
Notify	The robot sends a notification with a message pre-specified by the user. The message can be sent across different FSAs and is received by the Notified trigger by matching the message string. It is useful when the user wants to create a mission which requires synchronization of two FSAs. This message will not be, however, sent to different robots. Use NotifyRobots (explained below) for communicating with other robots.	AuRA.naval AuRA.urban
NotifyGoals	The robot notifies its goal position to other robots.	AuRA.urban
NotifyRobots	This state works similar to how Notify works (explained above). However, in this state, the robot can pass the message to other robots, and received by the Notified trigger by matching the message string. It is useful when the user wants to create a mission in which one robot sends commands to or synchronizes with other robots.	AuRA.naval AuRA.urban
NotifyRobots_ObjectLocation	The robot broadcasts the location of the closest object.	AuRA.naval
PickUp	The robot picks up nearest object specified by the user.	AuRA.naval AuRA.urban
ProceedAlongHallway	The robot moves down the hallway. This state is assembled by “Move-to-goal”, “Stay-on-path”, and “Avoid-static-obstacles” schemas [1].	AuRA.urban
PutInEOD	The robot puts the object it is carrying into the EOD area.	AuRA.naval AuRA.urban
ResetWorld	Given the specified overlay, this task will terminate the current mission, and restart it from the beginning. It is useful for experiments that need to be repeated a number of times.	AuRA.naval AuRA.urban

STATE	DESCRIPTION	Supported by
Run_EODTask	The robot executes EOD Task.	AuRA.naval
Run_InspectTask	The robot executes Inspect Task.	AuRA.naval
Run_InterceptTask	The robot executes Intercept Task.	AuRA.naval
Run_ObserveTask	The robot executes Observe Task.	AuRA.naval
Run_TrackTask	The robot executes Track Task.	AuRA.naval
SLUAV_Transport	The robot transports a SLUAV.	AuRA.naval
SetCameraTrackerMode	The robot changes its camera tracking mode.	AuRA.urban
SetMobilityType	The robot changes the mobility type of the robot. Available robot types are: UGV, UUV, USV, UAV, SLUAVU, and SLUAVL. Also, if UXV is selected, both UGV and USV will be supported depending on the terrain.	AuRA.urban
SetRobotColorId	This is an initialization state, which sets the unique identifier (the color) of the robot for Line Of Sight behaviors (states with the LOS prefix). The color identifier which is assigned to each robot must correspond to the actual color of the robot since, in simulation, the vision system of the robot recognizes different teammates on the basis of their color. This color can be set when running the mission from cfgedit or by modifying the CMDL file which is given as an input to the mlab command.	AuRA.urban
Standby*	The robot stands by to proceed the mission after checking its motor. The motor test failure will be notified to the user. This state has nested sub-FSA states. See the description for the SurveyRoom* task below regarding the “*” notation.	AuRA.urban
StartSubMission	This task will start a segment of the mission that considered to be a sub-mission (Note: it is not “sub-FSA”). This task should be used with SubMissionReady trigger as a pair. If SubMissionReady is set to be “DEPLOY_FIRST”, StartSubMission will make the robot to go to the designated location. If SubMissionReady is set to be “EXECUTE_IMMEDIATELY”, the state immediately after SubMissionReady will be executed (i.e., the robot will not go to the deployment position). Upon storing the mission plan, <i>CBRServer</i> will treat StartSubMission as a marker to indicate the break point of splitting the mission plan in useful pieces.	AuRA.naval AuRA.urban
StayWith	The robot stays with this object.	AuRA.naval AuRA.urban
Stop	The robot stops moving.	AuRA.naval AuRA.urban

STATE	DESCRIPTION	Supported by
SurveyRoom*	As explained in the TaskExited section above, SurveyRoom* is an example of states with nested sub-FSA states and triggers (Figure 40). Assuming that the robot is in a room, it looks for a possible object (e.g., biohazard). If the robot finds the possible biohazard, it will conduct a test to identify whether the object is a real biohazard or not. The robot will exit this task if the object was a real biohazard. If the object was not a real biohazard, the robot will keep looking for another possible biohazard. The robot will also exit the task if it determines that there is no possible biohazard in the room. In the <i>MissionLab</i> system, the “*” at the end of a state name, like SurveyRoom* or WanderRoom* , denotes that this state contains at least one TaskExited state in its nested sub-FSA states and triggers, and when the user chooses this state from the menu, the TaskExited trigger will be automatically generated to match with the TaskExited . SurveyRoom* is composed with the following nested sub-FSA states and triggers: Start , Immediate , LookFor , Detect , NotDetected , MoveToward , Near , TestObject , TestNegative , TestPositive , and ExitTask .	AuRA.urban
Talk	The robot reads and talks specified message.	AuRA.urban
Telop	It enables the Teleautonomous Operation (Telop). You can drive the robot with a joystick-like window.	AuRA.naval AuRA.urban
Terminate	The robot will terminate its own process.	AuRA.naval AuRA.urban
TerminateObject	The robot terminates the nearest (red) object.	AuRA.naval AuRA.urban
TerminateMission	The entire mission will be terminated.	AuRA.naval AuRA.urban
TestObject	The robot performs a test to identify the object. The user can use the TestPositive or TestNegative trigger (both explained below) to get out from the state and move on to next one.	AuRA.naval AuRA.urban
TrackTarget	The robot follows the target while taking a certain distance.	AuRA.naval
UAV_Move_To_Goal	The robot moves to a specified location assuming its mobility is UAV.	AuRA.naval
UAV_Move_To_Notified	The robot moves to a notified location assuming its mobility is UAV.	AuRA.naval
UnmarkDoorway	The robot unmarks the nearest doorway.	AuRA.urban
UpdateCSBSensorData	The robot updates the sensor data for the communication sensitive behavior.	AuRA.urban
WaitForProceed	The robot stops moving until proceeding mission is granted.	AuRA.naval AuRA.urban
Wander	The robot moves in random directions.	AuRA.naval AuRA.urban
WanderRoom*	This state also contains nested sub-FSA states and triggers. Assuming that the robot is in a room, the robot tries to moves randomly while it tries to maintain being inside the room, until a timer expires. WanderRoom* is composed of the following sub-FSA states and triggers: Start , Immediate , Wander , InRoom , EnterRoom , InHallway , Wait , and ExitTask .	AuRA.urban

Table 2: Available Triggers in *MissionLab*

TRIGGER	DESCRIPTION	Supported by
AboutFaceCompleted	The transition occurs when the robot faces in the opposite direction.	AuRA.naval AuRA.urban
Alerted	The transition occurs when the alert message has sent.	AuRA.naval AuRA.urban
AtDoorway	The transition occurs when the robot detects a doorway.	AuRA.urban
AtEndOfHall	The transition occurs when the robot reaches the end of the hallway.	AuRA.urban
AtGoal	The transition occurs when the robot is at the specified location. You can also specify a tolerance of which how far the robot can be off from the goal location.	AuRA.naval AuRA.urban
AtGoalInFormation	The transition occurs when a group of robots is at the specified location, and forming a specified formation.	AuRA.urban
AtOrSkipGoal	The transition occurs when the robot is at goal, or asked to skip goal by the user.	AuRA.urban
AwayFrom	The transition occurs when the robot is away from specified objects.	AuRA.naval AuRA.urban
ClearFlag	The transition occurs immediately, and this trigger clears the named flag to false.	AuRA.urban
CNP_AuctionReady	The transition occurs when the robot has a task for bid.	AuRA.naval
CNP_IsAuctionEnded	The transition occurs when the auction ended.	AuRA.naval
CNP_IsAuctionFailed	The transition occurs when the auction failed.	AuRA.naval
CNP_LostTask	The transition occurs when the robot loses a task.	AuRA.naval
CNP_TaskChanged	The transition occurs when the task constrains change.	AuRA.naval
CNP_TaskCompletionNotified	The transition occurs when completion of the CNP task is notified.	AuRA.naval
CNP_TaskRenegingNotified	The transition occurs when reneging of the CNP task is notified.	AuRA.naval
CNP_WonTask	The transition occurs when the robot wins a task.	AuRA.naval
CNP_WonTaskIs	The transition occurs when the robot won a specified-task.	AuRA.naval
Detect	The transition occurs when the robot detects the object specified.	AuRA.naval AuRA.urban
DetectAlternateHallway	The transition occurs when the robot detects a intersection of hallways.	AuRA.urban
DetectFriendlyRobots	The transition occurs when the robot detect a robot which is considered as a teammate. You must specify in the parameters the color of the robots which should be considered as teammates.	AuRA.urban
DetectMotivated	The transition occurs when the robot sees an object and all motivational variables have values that fall in the intervals specified by the corresponding upper and lower thresholds for each variable.	AuRA.urban
DetectSound	The transition occurs when the robot hears a sound having higher volume than the volume.threshold.	AuRA.urban

TRIGGER	DESCRIPTION	Supported by
DetectSoundMotivated	The transition occurs when the robot hears a sound and is suitably motivated. This trigger combines sound capabilities and motivational variables (Section 3.1.12). When the current curiosity level of the robot is above a given threshold, it behaves like DetectSound mentioned above. Moreover, every time this trigger gets activated, the curiosity value of the robot is decreased by 0.10. This effectively makes the robot indifferent to sounds after a while (habituation).	AuRA.urban
FlagIsClear	The transition occurs when the flag is not set.	AuRA.urban
FlagIsSet	The transition occurs when the flag is set.	AuRA.urban
Holding	The transition occurs when the robot is holding the specified object.	AuRA.naval AuRA.urban
Immediate	The transition occurs immediately without any condition.	AuRA.naval AuRA.urban
InAlternateHallway	The transition occurs when the robot enters the alternate hallway.	AuRA.urban
InHallway	Transition occurs when the robot is within a hallway.	AuRA.urban
InRoom	Transition occurs when the robot is within a room.	AuRA.urban
IsFacing	The transition occurs when the robot is facing a specified compass direction.	AuRA.naval AuRA.urban
MarkedDoorway	The transition occurs when the nearest doorway has been marked.	AuRA.urban
MessageSent	The transition occurs immediately after a message has sent.	AuRA.naval AuRA.urban
MovedDistance	The transition occurs when the robot has moved a desired distance.	AuRA.naval AuRA.urban
Near	The transition occurs when the robot is near specified objects.	AuRA.naval AuRA.urban
NearNotifiedObject	The transition occurs when the robot is near the notified object.	AuRA.naval
Never	The transition does not occur. It may be useful for debugging.	AuRA.naval AuRA.urban
NoProgress	The transition occurs when the robot is stuck and making no progress moving towards the specified goal.	AuRA.naval AuRA.urban
NotAtEndOfHall	The transition occurs when the robot is not near the end of the hallway.	AuRA.urban
NotDetected	The transition occurs when the object once detected by the robot is no longer in sight.	AuRA.naval AuRA.urban
NotHolding	The transition occurs when the robot is not holding the specified object.	AuRA.naval AuRA.urban
Notified	The transition occurs when the robot receives a message that was sent by the Notify state or the NotifyRobot state (both explained above).	AuRA.naval AuRA.urban
Notified_ObjectLocation	The transition occurs when the robot is notified about object location.	AuRA.naval
Proceed	The transition occurs when the console commands to proceed the mission.	AuRA.naval AuRA.urban
SetFlag	The transition occurs immediately, and this trigger sets the named flag to true.	AuRA.urban

TRIGGER	DESCRIPTION	Supported by
SkipGoal	The transition occurs when "skip waypoint" button is pressed in console.	AuRA.urban
StrongCommSignal	The transition occurs when the robot has the communication signal above the threshold.	AuRA.urban
SubMissionReady	See StartSubMission task in Table 1.	AuRA.naval AuRA.urban
TaskExited	As it was explained in the ExitTask section above, the transition occurs when the robot is in the ExitTask state of the sub-FSA whose message string matches with the one specified for this trigger.	AuRA.naval AuRA.urban
TelopComplete	The transition occurs when the <i>Telop</i> window is closed.	AuRA.naval AuRA.urban
TestNegative	The transition occurs when the result of the TestObject state (explained above) is negative.	AuRA.urban
TestPositive	The transition occurs when the result of the TestObject state (explained above) is positive.	AuRA.urban
ThroughDoorway	The transition occurs when the robot passes through a doorway.	AuRA.urban
UnmarkedDoorway	The transition occurs when the nearest doorway has been unmarked.	AuRA.urban
Wait	The transition occurs when the timer expires.	AuRA.naval AuRA.urban

by mistake, and right clicking exits *mlab*'s waypoint mode. If you wish, you can save the waypoints in an overlay file. An example constructed FSA with the waypoints tool and the executed waypoints mission are shown in Figure 44 and 45, respectively.

3.3.6 Using Path Planning Tool

In addition to the waypoint specification functionality described in Section 3.3.5, *CfgEdit* allows users to take advantage of a full scale path planner when specifying missions.

Path Planner

The path planner is a separate program invoked from *CfgEdit* when users press the **Path Plan** button located on the left side of the screen. Pressing the button opens a separate window that displays a map (Figure 46). The user must specify a start and end position for the robot by left clicking on the map. The planner then constructs a path between these two points. The user has the option of constructing another path by repeating the clicks or quitting the planner by middle clicking or right clicking on the map window.

The path generated by the planner consists of a series of line segments. After quitting, the planner saves the points forming the line segments in a file named "points.txt" (in the directory from which *CfgEdit* was invoked). This file serves as the interface between the planner and *CfgEdit*. *CfgEdit* reads that file and constructs an FSA similar to the one shown on Figure 47. This FSA can be compiled and executed as shown in Figure 48. Note, however, that *mlab* uses the overlay file to run the mission and not the map file that was used by the planner.

How the Planner Works

The code for the path planner can be described in six steps:

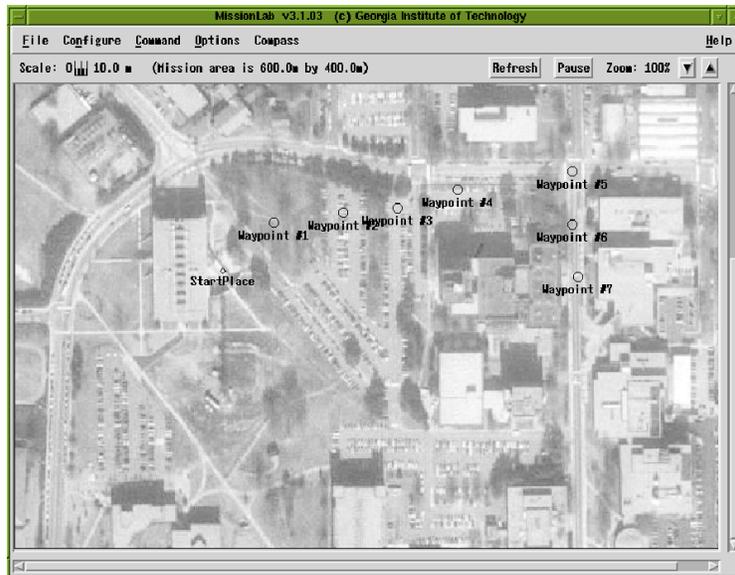


Figure 43: *mlab*: adding waypoints.

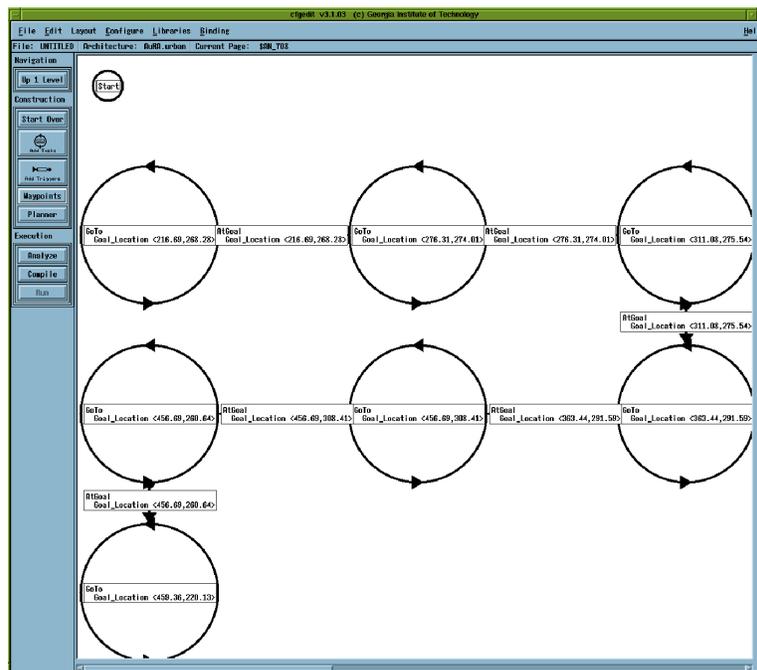


Figure 44: *CfgEdit*: constructed FSA with the waypoints function.

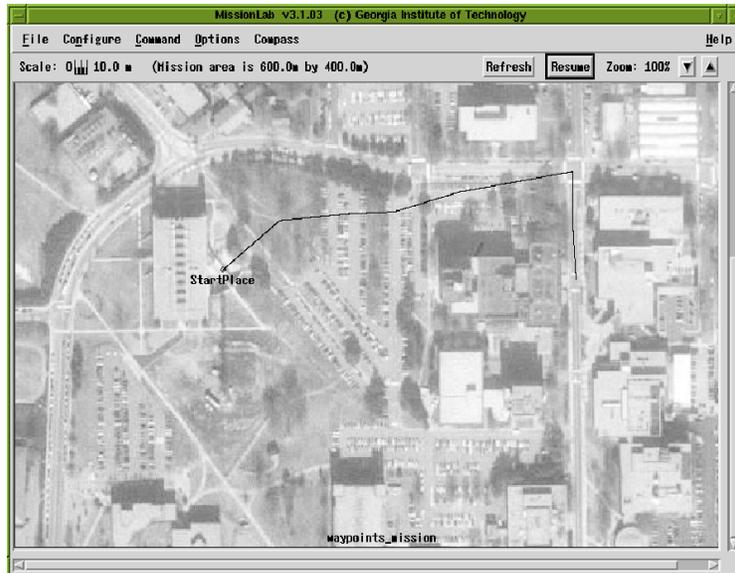


Figure 45: *mlab*: robot executing the waypoints mission.

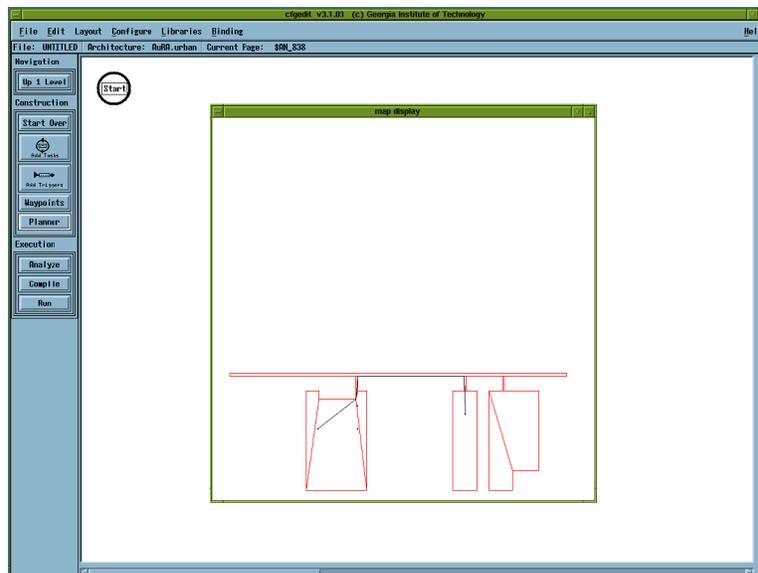


Figure 46: Path Planner Window.

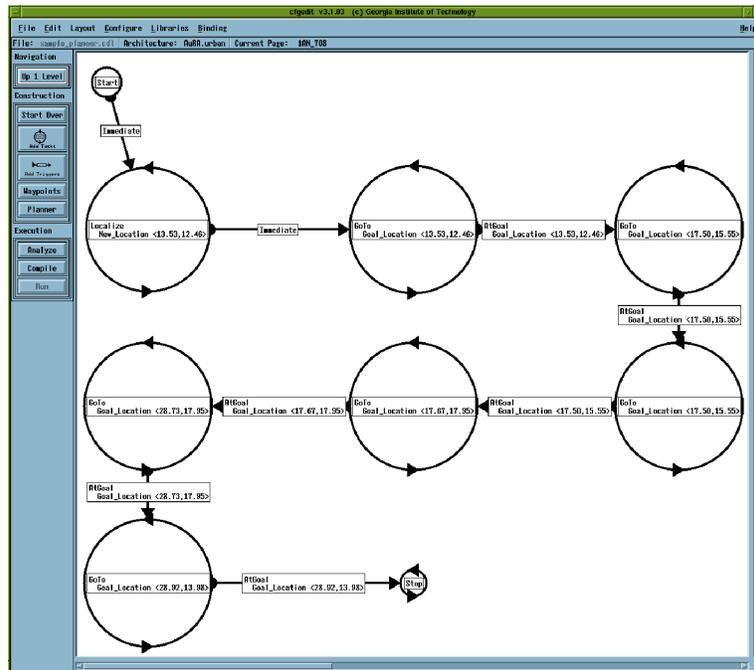


Figure 47: *CfgEdit*: generated FSA by the path planner.

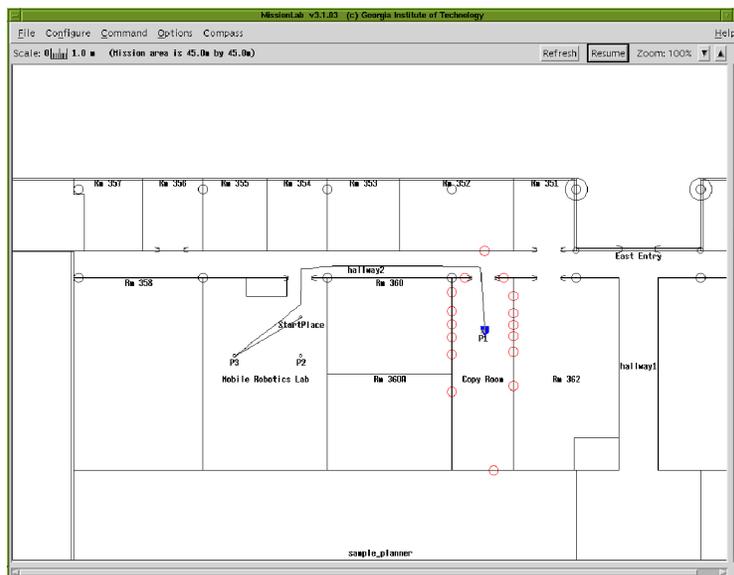


Figure 48: *mlab*: a robot following the path planned by the path planner.

1. Reads a description of the environment from a map file. This file defines the position of the walls and obstacles in the environment. The walls should form a closed polygon which we call the environment border. The format of this file is described below.
2. Before we do any form of path planning we have to apply some standard techniques from the robot motion planning literature. We define a configuration space by shrinking the border by the radius of the robot. From now on all operations are performed using the shrunk border only. This is the reason why the displayed map (Figure 46) may look strange.
3. The result from Step 2 is a complex polygonal region. To simplify the path planning task we partition this border into a set of convex polygonal regions. This explains why the map on Figure 46 has lines across the rooms.
4. The user is now prompted to specify start and end points for the robot path. This is done by asking the user to click twice (once for each point) in the window displaying the map.
5. The two points from Step 4 are passed as arguments to the path planning module and a path is constructed between them. The path consists of a sequence of line segments connecting the start and end points. The line segments do not cross the boundary or the obstacles at any point. The path planning module uses the A* algorithm to find the path.
6. Outputs a text file which specifies the path that the robot should follow. The format of this file is described below.

File Format for the Map Files (*.map)

The map file describes the environment in which the robot will be moving. The file is a regular text file with format as described below. The file has three sections: walls, obstacles, and landmarks. Each of the sections begins with a keyword and a number. The keyword is just the name of the section (in capital letters) and the number specifies how many lines for this section follow. Please note that the map file is similar to the overlay file but has different format. This is dictated by the specifics of the path planner that requires that all points on the map are specified in counterclockwise direction for walls, and clockwise direction for obstacles. Walls should form a closed non-intersecting polygon; the same is true for every obstacle.

- **WALLS:** The entries in the WALLS section specify points in the Cartesian plane which if connected form a closed polygonal contour. The points have to be specified in counter-clockwise order and must form a complete contour. The first point should NOT be repeated at the end of the list.
- **OBSTACLES:** The entries in the OBSTACLES section are similar to the entries in the WALLS section but they have to be specified in clockwise fashion. As before the first point should NOT be repeated at the end of the list. The obstacles section specifies only one obstacle. To add more obstacles define multiple obstacle sections and end with a line containing 'OBSTACLES 0'.
- **LANDMARKS:** The entries in the LANDMARKS section specify environment landmarks. Each landmark is specified on a separate line. Each line contains two numbers (x and y coordinates of the landmark) and a string (type of landmark or how the landmark is displayed on the screen). The currently supported strings are SQ- for square and PIE-for a pie-slice. This is still an experimental section of the path planner and in this release landmarks are not taken into account when generating the path.

Here is an example of a map file. The map that the planner generates from this file is shown in Figure 49. It differs from Figure 46 only by the obstacle in the middle of the room. Also note how the number, size and shape of the convex regions changes when the obstacle is added.

```

WALLS 32
3.75 17.3
16.85 17.3
16.85 16.2
14.3 16.2
14.3 17.1
11.67 17.1
11.67 5.5
19.29 5.5
19.29 17.1
18.32 17.1
18.32 17.3
28.08 17.3
28.08 17.1
26.91 17.1
26.91 5.5
30.67 5.5
30.67 17.1
29.55 17.1
29.55 17.3
32.09 17.3
32.09 17.1
30.67 17.1
30.67 5.5
34.4 5.5
34.4 7.5
37.12 7.5
37.12 17.1
33.56 17.1
33.56 17.3
40.0 17.3
40.0 18.95
3.75 18.95

OBSTACLES 3
15.0 11.0
15.0 9.0
16.0 8.0
OBSTACLES 0

LANDMARKS 7
34.0 6.5 SQ
13.0 15.1 SQ
14.0 14.1 SQ
38.5 18.0 PIE
15.5 12.0 PIE
8.5 17.3 PIE
28.5 17.5 PIE

```

Contents of the “points.txt” File Generated by the Planner ³

File points.txt

```

13.53 12.46 1.0 0.45 0.40 0.23 0.5
17.5 15.55 1.0 0.45 0.40 0.23 0.5
17.5 15.55 1.0 0.45 0.40 0.23 0.5
17.67 17.95 1.0 0.45 0.40 0.23 0.5
28.73 17.95 1.0 0.45 0.40 0.23 0.5
28.92 13.98 1.0 0.45 0.40 0.23 0.5

```

Each line describes one point. The first two numbers are the point’s x and y coordinates. The next four numbers specify values for the parameters of the GoTo assemblage (`move_to_location_gain`, `avoid_obstacle_gain`, `avoid_obstacle_sphere`, `avoid_obstacle_safety_margin`) which *CfgEdit* builds for each point. The last parameter is used By *CfgEdit* to set the value for the Goal_Tolerance parameter of the AtGoal trigger coming out of the GoTo state.

To summarize Each line contains 7 numbers corresponding to:

³The common user need not be concerned with this file it serves just as an interface between two programs that are likely to change often.

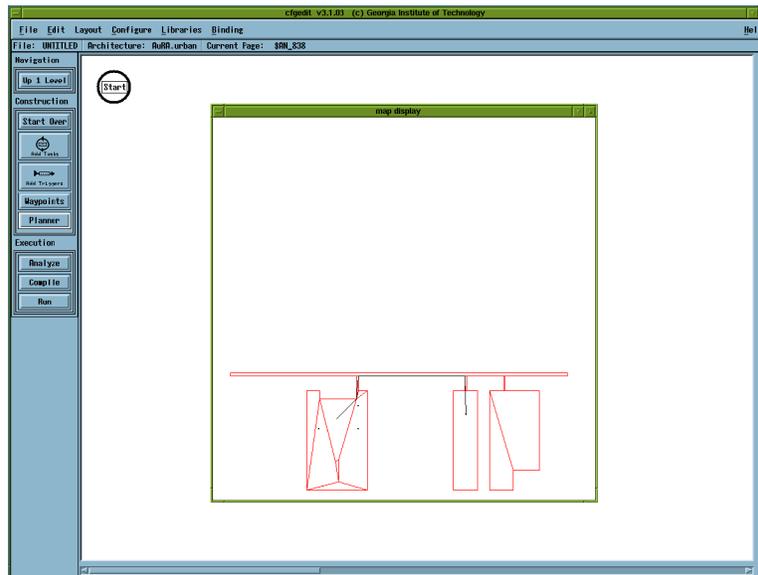


Figure 49: Path Planner Window: an obstacle exists in the middle of the room.

```
x
y
move_to_location_gain
avoid_obstacle_gain
avoid_obstacle_sphere
avoid_obstacle_safety_margin
Goal_Tolerance
```

An example demo script to launch the *CfgEdit* configured for Path Planner can be found in *[your MissionLab home]/demos/honda_demos/* (Run *demo_planner*).

Known Bugs

Currently there is no way to specify which *.map file to use with the planner when the planner is called from *CfgEdit*. It is hardwired in the code that the default file is “marc3f.map”. However, if the path planner is called without parameters it will prompt for a file name.

When running a mission generated with the planner the robot appears to jump from its initial point to the start point for the mission. This is due to the fact that the start position of the robot is specified in the default overlay file and the robot is drawn there before the mission is even started. Therefore, the first state of the FSA generated by the planner is “localize” which changes the robot’s position. The apparent jump has no effect on the mission other than leaving a robot trail as seen on Figure 48 .

3.3.7 Using Q-Learning for a Behavioral Selection

As you have seen so far, you can create a robot mission by assembling a sequence of FSA states (tasks) and triggers as you wish the robot to behave. Alternatively, *MissionLab* is also capable of assembling the tasks and triggers automatically during the run-time using reinforcement learning methods (Q-learning).

In order to create a robot mission that utilizes Q-learning, run *CfgEdit*, and descend the configuration level down to the third level of the configuration tree. By default, an FSA operator has been connected to the Wheel Actuator icon. After unlinking them by click on the button on the Wheel Actuator icon, delete the FSA operator using the “Delete” function. Choose the QLEARN coordinator from the list of operators

after click on **Operator** button (Figure 50). Connect the output of the QLEARN coordinator to the Wheel Actuator icon (Figure 51).

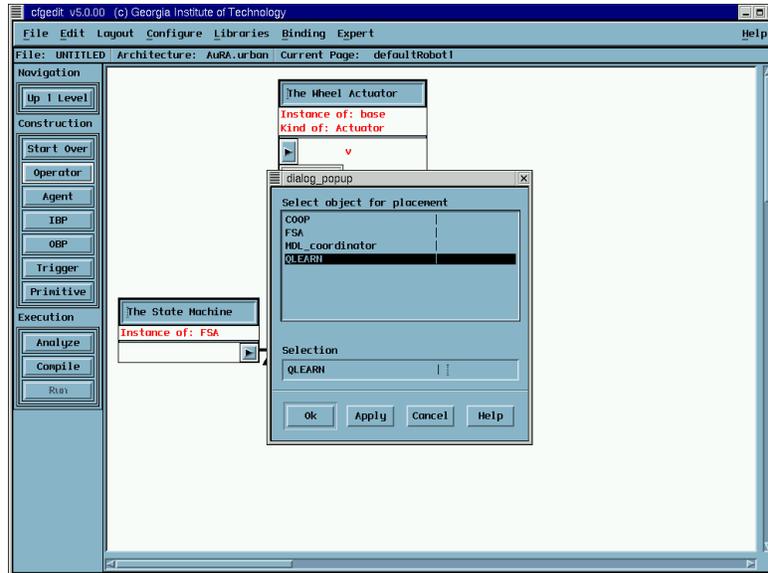


Figure 50: *CfgEdit* - QLEARN coordinator is selected from the operator list.

Unlike the FSA operator where you explicitly specified the sequence of the tasks and triggers in the fourth level of the configuration tree, all triggers and tasks that you wish the reinforcer to assemble should be connected to the QLEARN coordinator at this level. For example, to connect a new trigger to be assembled by the reinforcer, click on the **Trigger** button on the left, select the trigger (Figure 52), and place it in the workspace. After clicking on the **New** button under the word “triggers” in the QLEARN coordinator, link the arrow in the trigger and the arrow in the QLEARN coordinator.

Similarly, a task that you wish it to be assembled by the reinforcer can be connected to “society” section of the QLEARN coordinator. Clicking the **Agent** button on the left will popup the list of the tasks that can be used. Please note that if you try to connect more than one identical tasks to the QLEARN coordinator it may produce a linking problem during the compilation. In this case, click on the task icon to allow importing it from the library.

Each reinforcer $\langle state, action, reward \rangle$ tuple defines when the robot should be rewarded and by how much. This *state* is not the FSA state being mentioned in the previous sections. It is a binary state of the triggers (Table 3). For example, if there are three triggers (*InHallway*, *Detect* [Enemy], and *Near* [Mine]) are connected to the QLEARN coordinator from bottom to top, and only *InHallway* is true, then the reinforcer *state* is 100, or $(1 \times 2^2) + (0 \times 2^1) + (0 \times 2^0) = 4$. Thus, since there are three triggers in this case, there are 2^3 possible reinforcer *states*. The *action* section of the reinforcer defines which task should the robot be using when that reinforcer *state* of the triggers occurs. If the correct reinforcer *state* occurs, but the wrong *action* (task) is being performed, then the robot is not given the specified *reward*.

To add a reinforcer, click on the **New** button below the word “reinforcers”. It will create a new input point (shown as an arrow). Select “Toggle input between constant and link” from the “Layout” menu at the top, and click on the new arrow. This will convert the arrow into a text box. Type in the desired tuple in the text box.

Finally, to use the QLEARN coordinator in *CfgEdit*, a QfileName and ActionTimeout have to be entered in the appropriate text-boxes. The QfileName specifies the file in which to save the Qtable. Note that the filename has to have quotes around in when placed in the text-box (i.e., “Qfile”, not Qfile). The ActionTimeout is how many time-steps the algorithm will wait for something to happen before updating

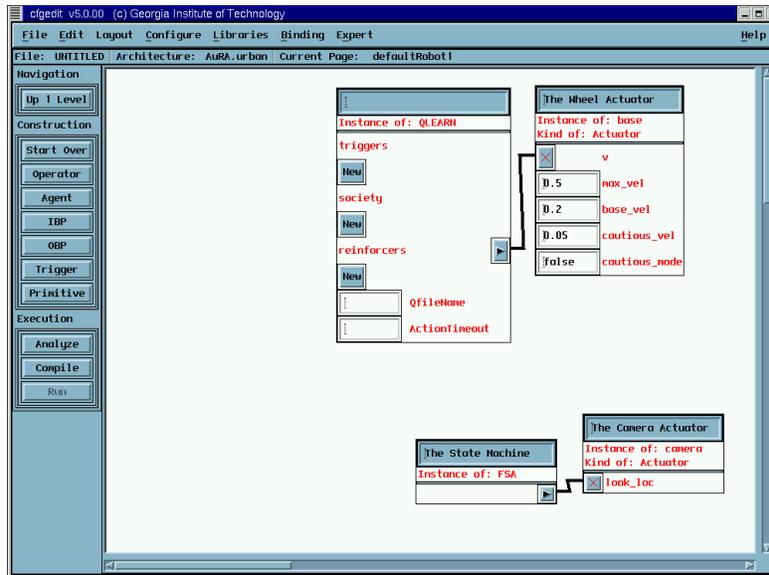


Figure 51: CfgEdit - QLEARN coordinator is connected to the Wheel Actuator icon.

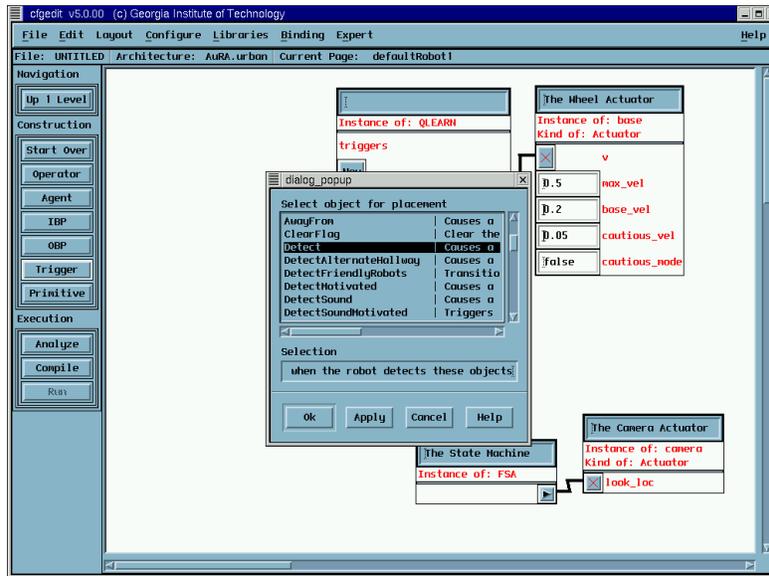


Figure 52: CfgEdit - One of the triggers is selected.

C	B	A	State
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

Table 3: Given three triggers from top to bottom, A B C, the table shows the resulting state

the Qtable. If the reinforcer *state* does not change, or if the *reward* is not given before ActionTimeout (in time-steps) have occurred, then the algorithm will query the Qtable and update its memory.

An example configuration using the QLEARN coordinator is shown in Figure 53. The identical CDL file (sample_qlearn.cdl) can be found in [your MissionLab home]/demo/mars_demos. In this configuration, the algorithm will be rewarded 10 points when it is *Near* [mine] AND *Detect* [Flags]. To experiment, run the robot with sample_qlearn.ovl in the same directory.

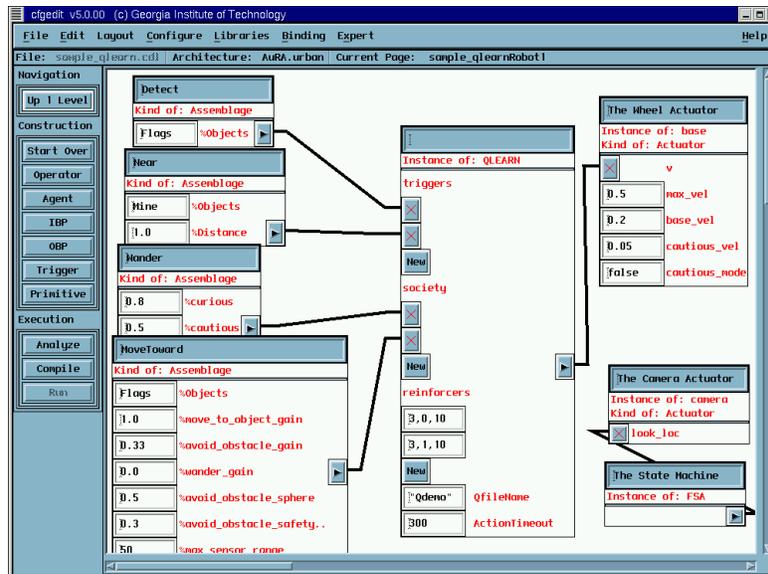


Figure 53: *CfgEdit* - Sample Configuration for Q-Learning.

3.3.8 Analyzing and Compiling the Robot Executable

Once you create a mission with the FSA diagram, and if you are comfortable with it, you can compile it to make a robot executable. To compile it, you can simply left-click the **Compile** button in the left menu bar. In order to check the feasibility of the real time running, you have to have a version of *MetaH*, a real-time analysis program, which was developed by Honeywell, Inc. for the DARPA sponsored Tactical Mobile Robotics project. If you have Honeywell's *MetaH*, you can invoke it by left-clicking the **Analyze** button located above the **Compile** button. However, because of the copyright issue, Honeywell's *rta* is not included in the *MissionLab* package. Thus, the current **Analyze** button will invoke a dummy program that mimics Honeywell's *rta*, but the result will be, of course, meaningless.

3.3.9 Executing the Robot Executable

As you can see from Figure 15 on Page 14, upon compilation, *CfgEdit* takes the following steps:

- **CDL \Rightarrow CNL:** The CDL (Configuration Description Language) file that contains information of the mission you created will be bound to [your MissionLab home]/lib/agents.AuRA.naval file that contains all the definitions for the states and triggers, and the *cdl* code generator produce a CNL (Configuration Network Language) code. As default, the CNL code will be saved in /tmp directory.
- **CNL \Rightarrow C++:** As soon as the CNL code is generated, the *cnl* compiler will be invoked, and it will be translated into a C++ code.

- **C++ ⇒ robot executable:** At this point, the GNU C Compiler (`gcc`) will compile the C++ code. The hardware drivers library (`libhardware_drivers.a`) and the primitive behavior library (`libcnl.a`) as well as other libraries in `[your MissionLab home]/lib` directory will be also linked to the robot executable.

As soon as you finish compiling a robot executable, you can run it with `mlab`. If you left-click the **Run** button in the left menu bar, `CfgEdit` will invoke the `mlab` and the run dialog (Figure 54) will ask you whether you want to run it as a simulation or run on a real robot (via `HServer`). The following are the options you can specify in the run dialog:

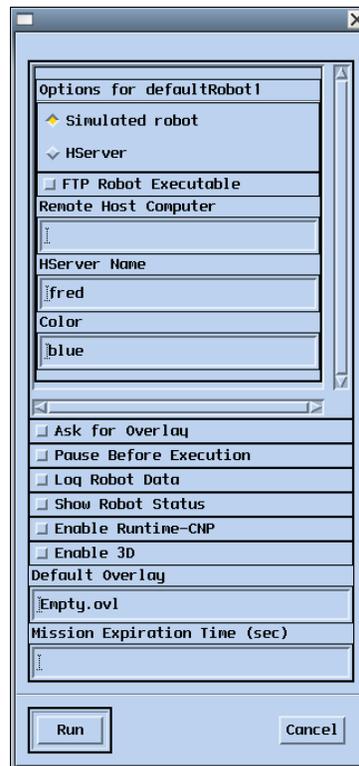


Figure 54: `CfgEdit` Run Dialog

- **Options for <robot name>:** This selection gives options to the users whether he or she wants to run the mission on a simulation (labeled “Simulated robot”) or a real robot (labeled “HServer”). These options are defined in the “.cfgeditrc” file (Section 3.3.1).
- **FTP Robot Executable:** If this check box is set, `cfgedit` will attempt to ftp the robot executable over to the computer listed under *Remote host computer* before executing *MissionLab*. With this option, `mlab` will look for the robot executable in the directory, specified as *RemoteHostRobotDir* in the “.cfgeditrc” file, on the remote machine, then, execute it. In other words, the robot executable has to be previously saved in that directory prior to execution. `CfgEdit` is, therefore, equipped with the capability of transferring the binary via ARPANET File Transfer Program (FTP). After pressing the **Run** button, `CfgEdit` will login to the remote machine as the user who are specified as *RemoteShellUserName* in the “.cfgeditrc” file and start downloading the file into *RemoteHostRobotDir* directory. In order for FTP to check the password automatically, you also have to create a file called “.netrc” in your home directory, and specify the password for the user *RemoteShellUserName*. If your remote machine is, for example, “cartman.cc.gatech.edu”, and your *RemoteShellUserName* is “demo”, your “.netrc” file should have the following lines:

```
machine cartman.cc.gatech.edu
login demo
password <password for user demo>
```

Since *mlab* uses *rsh* (remote shell) to execute the robot executable, you need to create the “.rhosts” file in the home directory of the *RemoteShellUserName*. on the remote machine as well. For example, if your username is “wendy” and your local machine is “stan.cc.gatech.edu”, write this line in the “.rhosts” file:

```
stan.cc.gatech.edu wendy
```

- **Remote host computer:** If this field is empty the robot executable is executed on the current machine. If this field is filled the robot executable is executed on the remote host via an rsh. Default is set by “.cgeditrc” option “DefaultHost =”.
- **Robot name:** This name is used as an ID for *HServer* if the user chooses to run the mission on a real robot.
- **Color:** The color specified in this field will be the color of the robot showing up in the *mlab* console. You can choose any rgb color name which is supported by your machine (Choose a color which has no space within its name). However, due to a bug, if the display of the *mlab* console is exported on PC or SGI monitors, the robot color may be different from what you specified. If you export the display to Sun monitors, the color of the robot should appear appropriately.
- **Ask for Overlay:** If this check box is set, *mlab* will bring up a file dialog box asking for an overlay. If unchecked, *mlab* will use the overlay listed under **Default Overlay**.
- **Pause Before Execution:** If this check box is set *mlab* will bring up a dialog box pausing the execution of the mission until the dialog box button is pressed.
- **Log Robot Data:** If this option is selected, the robot logs its runtime data. See Section 3.1.20 for more details on the robot data logging.
- **Show Robot Status:** If this option is selected, the status of the robot (showing its current state) will be displayed.
- **Enable Runtime-CNP:** If the mission contains the runtime-CNP behaviors and this option is turned on, the robots will attempt to bid for tasks during the execution.
- **Enable 3D:** If the overlay contains the 3D information, accompanied by the height map (Section 3.1.24), the *mlab* will display the robot and environment in 3D.
- **Default Overlay:** The overlay file that *mlab* will use unless **Ask for Overlay** box is checked. Default is set by “.cgeditrc” option “DefaultOverlay =”. (Remember to have the default overlay in the directory where you are running *CfgEdit* from.)
- **Mission Expiration Time (sec):** If the user wishes to force *mlab* to terminate the mission and quit its process after a certain time, the expiration time in seconds can be specified in this box. It can be useful for conducting a repetitive experiment.
- **Run:** Pressing this button will cause *cgedit* to first write out a *cmdl* file (Section 3.6) in the current directory. Usually, the *CMDL* file will be named after the name of the *CfgEdit* configuration that the user specified with **Save As** option (without “.cdl”). If the user hasn’t saved the configuration, the *CMDL* file will be named after the default configuration name specified by “.cgeditrc” option “DefaultConfiguration =”. *CfgEdit* will then execute (fork) *mlab*, starting the mission. If the user wishes, the same mission can be executed directly from a command line:

```
mlab -r <CMDL file name>
```

- **Cancel:** Pressing this button closes the dialog box without running the mission or saving the running options to a *CMDL* file.

3.3.10 Adding New States and Triggers

This section is for an advanced user, and describes how to add new states and/or triggers to the list that you use to create a mission. If you go through the following example steps, you can implement the “swirl” behavior into a new state.

1. **Choose Primitive Behaviors to Bind.** You have to have primitive behaviors to which your new state/trigger will incorporate. The primitive behaviors are written in the Configuration Network Language (CNL) and stored in the `[your MissionLab home]/src/libcni` directory. You can use those existing ones, or, if you wish, you can create your own primitive behaviors. If you are going to create your own primitive behaviors, make sure that you declare those in `[your MissionLab home]/src/libcni/cni.inc`. Programming in CNL is explained in Section 3.5, and, for details, please read the separate manual. Here, let us use the existing `MOVE_TO.cni` (the “Move-to-goal” behavior) and `SWIRL_STATIC_OBSTACLES.cni` (the “Swirl” behavior) that will be incorporated into our new state “GoTo_with_Swirl”. If you choose to use the existing ones, make sure that each CNL code has “procedure Vector” instead of “procedure bool”: “procedure Vector” is for a “state” while “procedure bool” is for a “trigger”. For example, in `SWIRL_STATIC_OBSTACLES.cni`, the procedure is specified as:

```
procedure Vector SWIRL_STATIC_OBSTACLES with
```

2. **Declare Inputs to the Primitive Behaviors.** Once you decide which primitive behaviors to use, you have to declare the inputs to those behaviors, if they have not been declared, in the “default.gen” and “default.AuRA.naval” files. The “default.gen” and “default.AuRA.naval” files are for a generic robot and for the Autonomous Robotics Architecture (AuRA) robot⁴, respectively, saved in the `[your MissionLab home]/src/cdl_code` directory, and also linked from for the `[your MissionLab home]/lib` directory. Since `MOVE_TO.cni` had been already used by the “GoTo” state, you can find in “default.gen” and “default.AuRA.naval” that the inputs for the behaviors are defined as:

```
// In default.gen
defAgent displacement MOVE_TO(
    boolean have_a_goal,
    location goal_rel_loc);

// In default.AuRA.naval
defAgent[AuRA.naval] binds MOVE_TO Vector MOVE_TO(
    boolean have_a_goal,
    Vector goal_rel_loc);
```

However, since this is the first time that `SWIRL_STATIC_OBSTACLES.cni`⁵ is to be used by any state, we have to declare its inputs. If you look at the `SWIRL_STATIC_OBSTACLES.cni` file, you will find that the “Swirl” behavior takes six inputs: “sphere”, “safety_margin”, “open_space”, “open_sphere”, “readings”, and “goal_rel_loc”:

```
// In SWIRL_STATIC_OBSTACLES.cni
procedure Vector SWIRL_STATIC_OBSTACLES with
    double sphere ;
    double safety_margin;
    double      open_space;
```

⁴We used to use “default.AuRA” and “agents.AuRA” as our primary libraries in the past. We are now mainly using “default.AuRA.naval” and “agents.AuRA.naval”, which are refined version of the previous one, and targeted for naval usage. For urban usage use “AuRA.urban” by modifying the configuration file (.cgedit) and set `restrict_names` to be `AuRA.urban` and `DefaultConfiguration` to be `urban.cdl`

⁵Recently, it was also integrated into this system (*GoToOutdoorNavigation*). Therefore, you can skip this part.

```

    double      open_sphere;
    obs_array   readings;
    Vector      goal_rel_loc;
header
body

```

“sphere” and “safety_margin” are the maximum and the minimum distances (in meters), respectively, that obstacles are reacted to by the swirl behavior; “open_space” is the minimum distance (in meters) that must be clear in a particular direction before it can be chosen as the heading by the swirl behavior; “open_sphere” is the maximum distance (in meters) that obstacles are reacted to to determine the heading by the swirl behavior; “readings” is an array that contains locations of obstacles; and “goal_rel_loc” is a vector directed to the goal location. Thus, let us declare these inputs in the “agents.gen” and “agents.AuRA.naval”, respectively, as the following:

```

// Copy this in default.gen
defAgent displacement SWIRL_STATIC_OBSTACLES(
    const number sphere,
    const number safety_margin,
    const number open_space,
    const number open_sphere,
    location goal_rel_loc,
    obs_array readings);

// Copy this in default.AuRA.naval
defAgent[AuRA.naval] binds SWIRL_STATIC_OBSTACLES Vector SWIRL_STATIC_OBSTACLES(
    const double sphere = {3.0},
    const double safety_margin = {0.5},
    const double open_space = {0.5},
    const double open_sphere = {0.5},
    Vector goal_rel_loc,
    obs_array readings);

```

3. **Construct Behavioral Assemblages.** In *CfgEdit*, primitive behaviors are coordinated and assembled as states using the cooperative coordination operators (*COOP.cnl*), and they are defined in the files “agents.gen” and “agents.AuRA.naval”. In our case, let us define the assemblages for the “GoTo_with_Swirl” state as the following, and paste it into the files. Probably, the best places to paste it into the files are just after the “GoTo” state is defined:

```

// Copy this in both agents.gen and agents.AuRA.naval
//*****
GoTo_with_Swirl:[
    %Goal_Location = {1.0, 1.0},
    %move_to_location_gain = {1.0},
    %swirl_obstacle_gain = {1.0},
    %swirl_obstacle_sphere = {3.0},
    %swirl_obstacle_safety_margin = {0.5},
    %swirl_obstacle_open_space = {0.5},
    %swirl_obstacle_open_sphere = {0.5},
    COOP(
        %Goal_Location = {^},
        %classes = {0}, // Will avoid everything.
        %swirl_obstacle_sphere = {^},
        %swirl_obstacle_safety_margin = {^},
        %swirl_obstacle_open_space = {^},
        %swirl_obstacle_open_sphere = {^},
        %max_sensor_range = {1000},

```

```

    members[A]<10,10> = $MoveToGoal,
    members[B]<10,130> = $Swirl_Obstacles,
    weight[A] = {^Range_01 %move_to_location_gain},
    weight[B] = {^Range_01 %swirl_obstacle_gain}<350,10>|GoTo and Swirl|
]<10,10>|Move to the specified location with swirling obstacles|
//*****

```

Notice that this assemblage is similar to the one defined in the “GoTo” state, except that “GoTo_with_Swirl” takes “\$Swirl.Obstacles” as the member[B] instead of “\$Avoid.Obstacles”, and there is no “\$Telop”. The “\$” sign in front of the function name indicates that this function is defined within the file, and calls the corresponding primitive behavior. For example, if you look at “agents.gen”, or “agents.AuRA.naval”, you will find the definition for “\$MoveToGoal” because its “Move-to-goal” behavior was being used by the “GoTo” state. The weight for each behavior is specified by the line “weigh[*] =”, and “^Range.01” means this value can be modified by a slider bar. The parameters specified in the lines above “COOP(...)” are the ones that you are going to be able to modify from the *CfgEdit* window (by middle-clicking the state in the FSA diagram). Their default values are specified in the braces {}. Each parameter then has to be specified within “COOP(...)” again, to be carried into the primitive behaviors. “{^}” indicates that the value is pushed up from higher levels. You may also want to note that the numbers specified as “<n1, n2>” are coordinates that are used by the *CfgEdit* window to calculate the position of the items. Now, let us define our “\$Swirl.Obstacles” as the following, and paste it in the same files, just after “\$MoveToGoal” is defined.

```

// Copy this in both agents.gen and agents.AuRA.naval
//*****
$Swirl_Obstacles:[
    %Goal_Location = {^},
    %swirl_obstacle_sphere = {^},
    %swirl_obstacle_safety_margin = {^},
    %swirl_obstacle_open_space = {^},
    %swirl_obstacle_open_sphere = {^},
    %max_sensor_range = {^},
    SWIRL_STATIC_OBSTACLES(
        %Goal_Location = {^},
        %max_sensor_range = {^},
        sphere = {^Distances_10 %swirl_obstacle_sphere},
        safety_margin = {^Distances_10 %swirl_obstacle_safety_margin },
        open_space = {^Distances_10 %swirl_obstacle_open_space},
        open_sphere = {^Distances_10 %swirl_obstacle_open_sphere},
        goal_rel_loc<10,50> = $GoalRelLoc,
        readings<10,81> = $ListOfObstacles
    )<393,26>|swirl obstacles|
]<66,36>|Swirl obstacles|
//*****

```

Again, the parameters specified in the lines above SWIRL_STATIC_OBSTACLES(...) are pushed up from the previous COOP(...). Those values are then carried inside SWIRL_STATIC_OBSTACLES(...) to match with its inputs that you declared in “default.gen” and “default.AuRA.naval”.

4. **Experiment with the New State.** At this point, you have completed adding the new state. The following commands may help you to see if you have successfully implemented the state:

```

cd [your MissionLab home]
mkdir test
cd test
cp [your MissionLab home]/src/overlays/Empty.ovl .

```

```
cp [your MissionLab home]/src/overlays/BigObstacle.ovl .
iptserver &
cfgedit
```

Now, create a “back_and_forth”-robot as you did in Section 2.2.2, except that, this time, use the newly created “GoTo_with_Swirl” state instead of the “GoTo” state. If you left-click a state in the FSA diagram, you should see the “GoTo_with_Swirl” state in the list as it is shown in Figure 55. To pick the “Goal_Location” of the two points the robot is going back and forth, you may want to use the `Pick From Overlay` feature: pick the two points, “StartPlace” and “Goal”, in the “BigObstacle.ovl”, for example. After you change “move_to_location_gain” to be 0.1, and “swirl_obstacle_sphere” to be 10.0, your mission will look like the one in Figure 56. After compiling and executing the robot executable, if your robot leaves the trace that swirls around the obstacle (Figure 57), you have successfully added a new state.

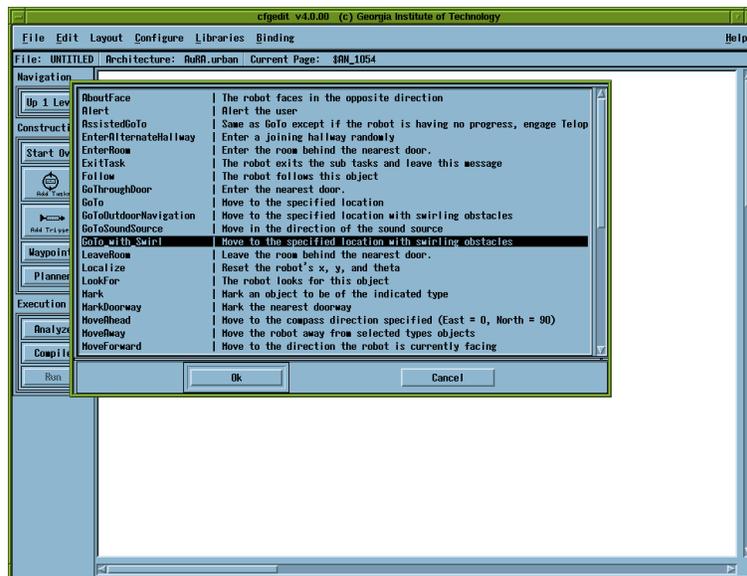


Figure 55: *CfgEdit* - the new state “GoTo_with_Swirl” created

The procedure for adding a new trigger is very similar to adding a new state that described above. However, for the trigger, make sure that you choose the CNL code that been called from the trigger has “procedure bool” instead of “procedure Vector”. Since it is not a vector, you do not make an assemblage using the cooperative coordination operator (*COOP.cnl*), neither.

3.3.11 Using Mission Expert

The Mission Expert functionality is explained in Section 3.9 (Page 104).

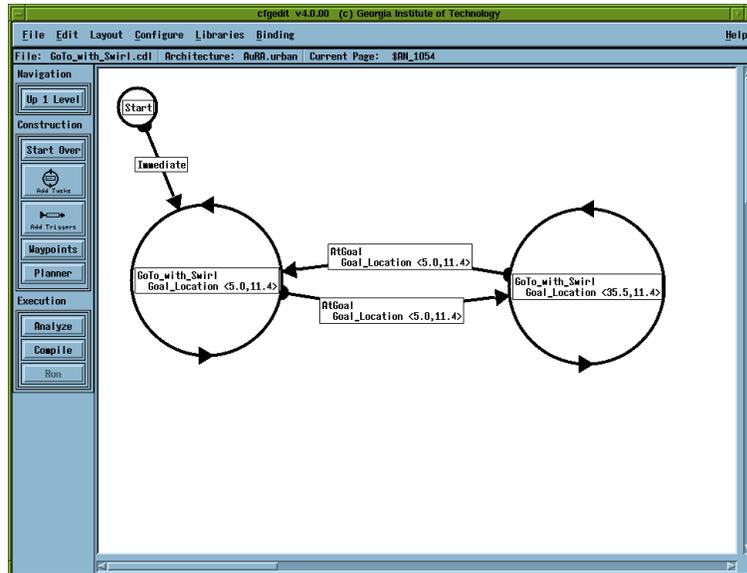


Figure 56: *CfgEdit* - a sample mission for the “GoTo.with.Swirl” state

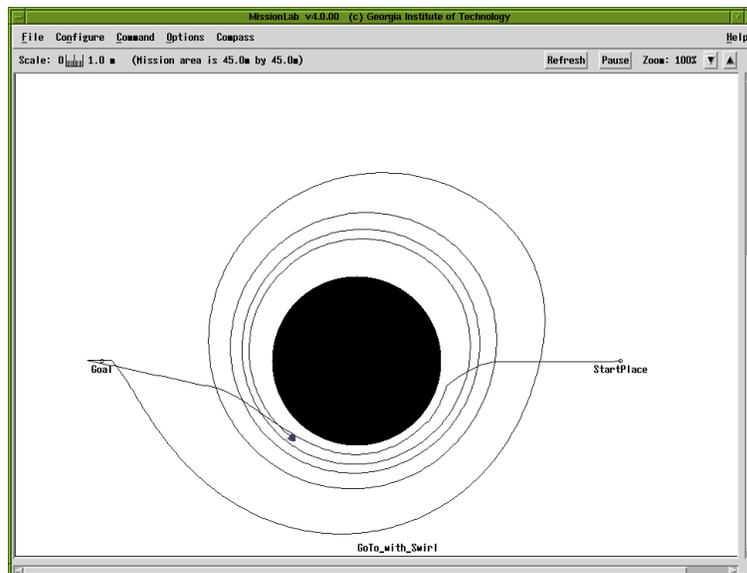


Figure 57: *mlab* - testing the “GoTo.with.Swirl” state

3.4 CDL (Configuration Description Language)

The configuration editor uses the Configuration Description Language (CDL) as its file representation. CDL is an agent-based programming language based on the Societal Agent theory, which presents a uniform view of all robot configuration components as agents. It uses a functional notation to support recursive construction of assemblages and named instantiation to support reusing components in multiple assemblages. Binding points are used in the configuration to denote connections to hardware devices. An explicit binding step is used to denote which sensors, actuators and robots are to be attached to the binding points. Code generators have been created which output CNL code for the AuRA architecture and SAUSAGES code for the UGV architecture. An example for a CDL representation of a configuration is shown below. Note that as the robot binding point *RobotBP* defines the configuration. The notation used to name the vehicle binding point is *instance_name:definition_name*, which specifies that this instance of *definition_name* is to be called *instance_name*. The parameter *bound_to* is used to note the binding of robot *stimp*, an instance of a MRV2 Denning robot. The FSA is specified as defining the robot. The states appear as instances of the *society* parameter and the transitions appear as the list of rules. A named instance of an agent used multiple places is created with the *instAgent* command (or *instOp* for operators). Notice that the wander skill is specified using this method since it is used in both the *look_for_can* and *look_for_basket* states in the FSA.

```

bindArch AuRA.naval;
instAgent drop_can from DETECT_LOST_CAN();
instAgent Denning_ultras from ultras:ULTRASONICS();

instOp Wander from COOP(
  weight[A] = {1.0},
  weight[B] = {0.8},
  weight[C] = {1.0},
  members[A] = noise:NOISE(
    persistence = {5.0},
    robot_heading = robot_heading:GET_HEADING(
      cur_pos = denning_encoders:xyt:SHAFTENCODERS()),
  members[B] = probe:PROBE(
    sphere = {2.0},
    safety_margin = {0.5},
    readings = Denning_ultras),
  members[C] = avoid_obs:AVOID_STATIC_OBSTACLES(
    sphere = {1.0},
    safety_margin = {0.5},
    readings = Denning_ultras));

RobotBP:vehicle(
  bound_to = Stimp:MRV2(
    trashbot_FSA:FSA(
      society[Start] = ,
      society[Look_for_can] = [ Wander ],
      society[Pick_up_can] = ,
      society[Look_for_basket] = [ Wander ],
      society[Put_can] = ,

      rules[Start] = if [true] goto Look_for_can,
      rules[Look_for_can] = if [detect_can:DETECT_CAN()] goto Pick_up_can,
      rules[Pick_up_can] = if [have_can:DETECT_HAVE_CAN()] goto Look_for_basket,
      rules[Look_for_basket] = if [detect_basket:DETECT_BASKET()] goto Put_can,
      rules[Put_can] = if [ true] goto Look_for_can,
      rules[Pick_up_can] = if [drop_can] goto Look_for_can,
      rules[Look_for_basket] = if [drop_can] goto Pick_up_can)))

```

The exact specification of the Configuration Description Language is defined below using grammar productions. Keywords printed in bold face such as **defArch** are terminal symbols. Punctuation and grouping characters such as “()[];” are also terminal symbols. The symbol ϵ denotes an empty production, that is, one that can match nothing. The remainder are nonterminals which are defined on the left side of a production. The main objects in CDL are the agents, coordination operators, binding points, and hardware devices. The remainder of the productions deal with architecture and binding related issues.

Start	Start DefineClass Start DefineBP Start InstAgent Start InstGroup Start InstSorA Start DefineRobot Start DefineOp Start DefineArch Start DefineType Start InstCoord Start InstBP Start InstRobot Start BindArch Start DefineSandA Start Agent DefineClass DefineBP InstAgent InstGroup InstSorA DefineRobot DefineOp DefineArch DefineType InstCoord InstBP InstRobot BindArch DefineSandA Agent
Agent	AGENT_NAME ROBOT_NAME GROUP_NAME BP_NAME RefRobot RefGroup RefCoord RefSorA RefBP ClsRef COORD_NAME
DefineArch	defArch NAME ;
BindArch	bindArch ARCH_NAME ; bindArch NAME ;
TypeSetArch	defType [ARCH_NAME] defType [NAME] defType
DefineType	TypeSetArch NAME ; TypeSetArch TYPE_NAME ;
DefineOp	OpSetArch TYPE_NAME NAME FSA_STYLE EndDefop OpSetArch TYPE_NAME NAME SELECT_STYLE EndDefop
EndDefop	(ParmDef) ; () ;
OpSetArch	defOp [ARCH_NAME] defOp
DefineRobot	RobotSetArch binds BP_CLASS NAME (BPList) ; RobotSetArch binds BP_CLASS NAME () ;
RobotSetArch	defRobot [ARCH_NAME] defRobot

BPList	BPList , BPparamdef BPparamdef
BPparamdef	sensor INLINE_NAME SENSOR_CLASS actuator INLINE_NAME ACTUATOR_CLASS
DefineBP	StartBP (ParmDef) ; StartBP () ;
StartBP	defIBP TYPE_NAME NAME defIBP NAME NAME defOBP TYPE_NAME NAME defOBP NAME NAME defRBP NAME
SensorSetArch	defSensor [ARCH_NAME] defSensor
StartSensor	SensorSetArch binds BP_CLASS TYPE_NAME NAME SensorSetArch binds BP_CLASS NAME NAME
ActuatorSetArch	defActuator [ARCH_NAME] defActuator
StartActuator	ActuatorSetArch binds BP_CLASS TYPE_NAME NAME ActuatorSetArch binds BP_CLASS TYPE_NAME ACTUATOR_CLASS ActuatorSetArch binds BP_CLASS NAME
SandA	StartSensor StartActuator
DefineSandA	SandA (ParmDef) ; SandA () ;
DefineClass	StartClass (ParmDef) ; StartClass () ;
AgentSetArch	defAgent [ARCH_NAME] defAgent
StartClass	AgentSetArch TYPE_NAME NAME AgentSetArch TYPE_NAME AGENT_CLASS
ParmDef	ParmDef , AParm AParm
AParm	TYPE_NAME NAME const TYPE_NAME NAME list TYPE_NAME NAME list const TYPE_NAME NAME
SorA	ACTUATOR_NAME SENSOR_NAME
Loc	i NUMBER , NUMBER i ϵ
InstAgent	instAgent Loc NAME from AGENT_CLASS ParmSet ; instAgent Loc NAME from SorA ParmSet ; instAgent Loc NAME from StartSorARef ParmSet ; instAgent Loc NAME from NAME ParmSet ;
ParmSet	(Glist) ()
Glist	Link Glist , Link
RobotParms	(RobotLinks) ()
RobotLink	Agent Link
RobotLinks	RobotLink RobotLinks , RobotLink
Link	LHS = Agent LHS = UP

	LHS = PU_INITIALIZER
	LHS = Rule
	LHS = INITIALIZER
	LHS =
	LHS = (RobotLinks)
LHS	PARM_NAME Loc
	PARM_NAME [INDEX_NAME] Loc
	PARM_NAME [NAME] Loc
	PARM_HEADER [NAME] Loc
	PARM_HEADER [INDEX_NAME] Loc
	PARM_HEADER
	PU_PARM_NAME Loc
MaybeAgent	Agent
	ϵ
Rule	if MaybeAgent goto INDEX_NAME
	if MaybeAgent goto NAME
MaybeName	INLINE_NAME
	ϵ
StartClsRef	MaybeName AGENT_CLASS
ClsRef	StartClsRef ParmSet Loc
InstRobot	StartRobotInst RobotParms
StartRobotInst	instRobot Loc NAME from ROBOT_CLASS
InstCoord	StartCoordInst ParmSet ;
StartCoordInst	instOp Loc NAME from COORD_CLASS
RefRobot	StartRefRobot RobotParms Loc
StartRefRobot	MaybeName ROBOT_CLASS
RefCoord	StartCoordRef ParmSet Loc
StartCoordRef	MaybeName COORD_CLASS
RefSorA	StartSorARef ParmSet Loc
SorAclass	ACTUATOR_CLASS
	SENSOR_CLASS
StartSorARef	MaybeName SorA Loc
	INLINE_NAME INLINE_NAME ACTUATOR_CLASS Loc
	INLINE_NAME INLINE_NAME SENSOR_CLASS Loc
	MaybeName SorAclass Loc
StartSorAInst	instActuator Loc NAME from ACTUATOR_CLASS
	instSensor Loc NAME from SENSOR_CLASS
InstSorA	StartSorAInst ParmSet ;
RefBP	MaybeName BP_CLASS ParmSet Loc
InstBP	instBP Loc NAME from BP_CLASS ParmSet ;
RefGroup	[AgentList] Loc
InstGroup	instGroup Loc NAME from [AgentList] ;
LinkorAgent	Agent
	Link
AgentList	LinkorAgent
	AgentList , LG

Consider the following example CDL configuration for a robot that simply moves forward until terminated by the operator. The first line binds the configuration to the AuRA architecture. The next line defines the vehicle binding point. This binding point is bound to a MRV2 robot named “stimpv” in the next line. The agent creating movement for the robot is an instance of the move_to_goal behavior named “mtg”.

Move_to_goal has two parameters, the relative location of the goal, and the maximum it must be to the goal before declaring success. In this case, the relative goal location is hard coded at 10 meters straight ahead. This will act as a carrot on a stick, keeping the robot moving north until it is terminated by the operator.

```
bindArch AuRA.naval;
move_forward:vehicle(
  bound_to = stimpy:MRV2(
    mtg:MOVE_TO_GOAL(
      goal_rel_loc<0,0> = {10,0},
      success_radius<0,0> = {1.0} )<150,95>
    )<317,69>
  )<214,64>
```

The Configuration Description Language is a powerful tool, aiding users in creating recursive specifications of multiagent configurations. The uniform representation of components at all levels of abstraction as agents simplifies object exchange and reuse. CDL is usable as a text-based language, but the real power comes from using the Configuration Editor to graphically create and maintain configuration specified in CDL.

3.5 CNL (Configuration Network Language)

When the configuration is bound to the AuRA architecture, the CDL compiler generates a Configuration Network Language (CNL) specification of the configuration as its output. CNL is a hybrid dataflow language[9] 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.

The use of communicating processing elements is similar to the Robot Schemas (RS)[13] 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.

CNL provides a separation between procedure implementations and specification of the dataflow graph's topology. CNL encourages this separation by allowing the procedures to be coded, tested and archived in standard Unix libraries. Instances of these procedures are specified in a separate step, thereby creating the dataflow network. The output of the CNL compiler is a C++ file created by merging the user's procedures and the compiler's data movement code specified by the node definitions. This output file can be compiled using any suitable ANSI C++ compiler targeted for the desired machine architecture. Using a standard programming language for the procedures simplifies converting existing code to CNL. The CNL language is documented in the Configuration Network Language User manual, included with this *MissionLab* package.

The code in Figure 58 is an example of the CNL procedure definition for the *avoid_static_obstacles* behavior used in the trash collecting (*trashbot*) configuration. There are four places that C++ code can occur in CNL: Bracketed by an **init/iend** pair, bracketed by a **once/header** pair within a procedure definition, bracketed by a **header/body** pair within a procedure definition, or bracketed by a **body/pend** pair within a procedure definition. Code within **init** blocks is not specific to a particular procedure. Code within **once** blocks is emitted before the body loop and therefore executes once on instantiation. Code within the **header** block is executed once each time the thread gains scope. For example, a node underneath a state in an FSA gains scope when that state becomes active and loses it when a new state takes over. Code within **body** blocks is surrounded with a while loop and executes once for each new set of input parameters. The predefined output parameter for the procedure is named *output*.

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. For example, both Figures 59 and 60 present a CNL representation of the *trashbot* configuration: Figure 59 shows the procedure generated to implement the FSA specified by the user, and Figure 60 shows the nodes created to implement the configuration in CNL. Each node in the configuration is an instantiation of a C++ function, forked as a lightweight thread using the C-Threads package[17] 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.

```

procedure Vector AVOID_STATIC_OBSTACLES with
  double sphere;
  double safety_margin;
  obs_array readings;
once
header
body
  VECTOR_CLEAR(output);

  for(int i=0; i<readings.size; i++)
  {
    double c_to_c_dist = len_2d(readings.val[i].center);
    double radius = readings.val[i].r + safety_margin;
    double mag = 0;

    if (c_to_c_dist ≤ radius )
    {
      // if within safety margin generate big vector
      mag = INFINITY;
    }
    else if (c_to_c_dist ≤ radius + sphere )
    {
      // generate fraction (0...1) how far are in linear zone.
      mag = (sphere - (c_to_c_dist - radius)) / sphere;
    }
    // otherwise, outside obstacle's sphere of influence, so ignore it

    if (mag ≠ 0)
    {
      // create a unit vector along the direction of repulsion
      Vector repuls;
      repuls.x = -readings.val[i].center.x;
      repuls.y = -readings.val[i].center.y;
      unit_2d(repuls);

      // Set its strength to the magnitude selected
      mult_2d(repuls, mag);

      // Add it to the running sum
      plus_2d(output, repuls);
    }
  }
pend

```

Figure 58: CNL code for avoid static obstacles behavior

```
#include "cnl.inc"
modulename = Stimp;

procedure Vector trashbot_FSA_proc with
  Vector agent_Look_for_basket;
  Vector agent_Put_can;
  Vector agent_Look_for_can;
  Vector agent_Start;
  Vector agent_Pick_up_can;
  boolean detect_basket;
  boolean drop_can;
  boolean detect_can;
  boolean have_can;
header
  const int Put_can = 4;
  const int Look_for_basket = 3;
  const int Start = 0;
  const int Look_for_can = 1;
  const int Pick_up_can = 2;
  int state = Start;
body
  switch(state)
  {
    case Look_for_basket:
      if(detect_basket) state = Put_can;
      else if(drop_can) state = Pick_up_can;
      output = agent_Look_for_basket;
      break;
    case Put_can:
      state = Look_for_can;
      output = agent_Put_can;
      break;
    case Look_for_can:
      if(detect_can) state = Pick_up_can;
      output = agent_Look_for_can;
      break;
    case Start:
      state = Look_for_can;
      output = agent_Start;
      break;
    case Pick_up_can:
      if(have_can) state = Look_for_basket;
      else if(drop_can) state = Look_for_can;
      output = agent_Pick_up_can;
      break;
  }
pend
```

Figure 59: The CNL procedure definition generated to implement the *trashbot* configuration

```

node trashbot_FSA is trashbot_FSA_proc with
  agent_Look_for_basket = Wander;
  agent_Put_can = NULL;
  agent_Look_for_can = Wander;
  agent_Start = NULL;
  agent_Pick_up_can = NULL;
  detect_basket = trig_detect_basket;
  drop_can = trig_drop_can;
  detect_can = trig_detect_can;
  have_can = trig_have_can;
nend

node trig_detect_basket is DETECT_BASKET with nend

node trig_drop_can is DROP_CAN with nend

node trig_detect_can is DETECT_CAN with nend

node trig_have_can is HAVE_CAN with nend

node Wander is COOP with
  weight = {1.0}; weight = {0.8}; weight = {1.0};
  members = noise; members = probe; members = avoid_obs;
nend

node noise is NOISE with
  persistence = {5.0};
  robot_heading = robot_heading;
nend

node probe is PROBE with
  sphere = {2.0};
  safety_margin = {0.5};
  readings = Denning_ultras;
nend

node avoid_obs is AVOID_STATIC_OBSTACLES with
  sphere = {1.0};
  safety_margin = {0.5};
  readings = Denning_ultras;
nend

node robot_heading is GET_HEADING with
  cur_pos = xyt;
nend

node Denning_ultras is ultras with nend

node xyt is SHAFTENCODERS with nend

```

Figure 60: The CNL nodes generated to implement the *trashbot* configuration in the AuRA architecture. Notice that the *trashbot.FSA_proc* is instantiated as a node.

3.6 CMDL (Command Description Language)

The command description file, or the CMDL file, contains the information necessary to set up a scenario and execute a series of commands. The structure and syntax of the command description file is explained in this section.

The command description file is organized into two parts (1) the background information part and (2) a list of commands to be executed. The basic syntax of the file is described below. White space (such as spaces, tabs, and blank lines) is ignored. Comments can be included on any line by using two dashes in a row; the rest of the line will be ignored.

In the following descriptions, items in italics would be replaced by appropriate values from the user.

3.6.1 Mission background information part

The first part of the file is a description of mission background information. It consists of a series of entries which define this information. The currently implemented entries are:

- **Mission name:** The user can specify the mission name with this entry. It has the following format:

```
MISSION NAME "the user's name for the mission"
```

- **Overlay:** This entry allows the user to indicate which file to read to load the overlay information. It has the following format:

```
OVERLAY "filename"
```

This command should be placed early in the file, since it may override some effects of the following commands.

- **Starting point:** Using this entry, the user specify a named starting position. This definition overrides any starting point specified in the overlay file (assuming they both have the same name). It has the following format:

```
SP Name x y
```

where *x* and *y* are the eastward and northward coordinates of the starting point (with respect any origin specified in any previously loaded overlay file).

- **Robot Definition:** The user can define new robots using this command. The format of the command is:

```
NEW-ROBOT Name executable-name [Color] [hostname] [( key1 = value1, key2 = value2, ... )]
```

where *Name* is the name of the robots to be used in subsequent unit descriptions, and *executable-name* is the filename of the robot executable program. The parameter "*Color*" is any legal X windows color name, *hostname* is the optional name of a host on which to run the robot program, and the optional list of key/value pairs. The key/value pairs are parameters to send to the robot to customize its behavior. The values should be quoted if they are strings with internal white space. All of this key/value parameter data is sent to the robot program's internal database after it has started executing. (Note: If you try to set a value and it does not seem to have the desired effect, double-check the spelling of the key and make sure it is correct.) It is then up to the robot program to use the parameter data.

There is one robot defined by default, **Robot**, which cannot be redefined. It is equivalent to the robot definition:

```
NEW-ROBOT robot "robot"
```

In the program "robot" included in this release⁶, numerous parameters can be given which control the operation of the robot's motor behaviors. See Table 4 for details. A detailed description of motor

KEYWORD	VALUE DESCRIPTION	Default
avoid_obstacle_sphere	Maximum distance at which the avoid_obstacle behavior begins to react to obstacles (in meters).	30.0
avoid_obstacle_safety_margin	Distance at which inter-robot collisions are assumed to occur (in meters).	5.0
base_velocity	Adjusts the overall speed of the robot (in meters/sec).	5.0
cautious_velocity	The top speed of the robot when in cautious mode (in meters/sec).	7.0
cautious_mode	Controls whether cautious_velocity is used in place of max_velocity . Value should be YES or NO.	NO
detect_obs_range	Maximum range at which the detectors can sense obstacles (in meters).	30
formation_spacing	Characteristic distance between robots in a formation (in meters).	5.0
formation_saturation_length	Distance (in meters) from the nominal formation position beyond which the robot's maintain-formation magnitude is set at maximum.	20.0
formation_dead_zone_radius	Distance (in meters) from the nominal formation position at which no further formation position correction is applied.	5.0
formation_slop	This is how far out of position the robot may be, yet still be considered in formation (in meters).	10.0
formation_default_trigger	Distance (in meters) from the goal at which a default heading is used for the formation reference instead of the goal heading.	100.0
max_velocity	Normal top speed of the robot (in meters/sec).	7.0
move_to_goal_success_radius	How close the robot needs to be to the goal before declaring success (in meters).	5.0
navigation_avoid_obstacle_gain	Weight applied to the avoid_obstacle behavior in navigate mode.	1.5
navigation_avoid_robot_gain	Weight applied to the avoid_robot behavior in navigate mode.	2.0
navigation_avoid_robot_min_range	If the robot is within this range of another robot (in meters), the repulsion vector from it is set at a maximum value.	15.0
navigation_avoid_robot_sphere	Other robots beyond this range (in meters) are not considered by the avoid robot schema.	30.0
navigation_formation_gain	Weight applied to the formation behavior in navigate mode.	1.0
navigation_move_to_goal_gain	Weight applied to the move_to_goal behavior in navigate mode.	0.8
navigation_noise_gain	Weight applied to the noise behavior in navigate mode.	0.1
navigation_success_radius	Distance the robot can stray from the desired formation position before a correcting force is applied (in meters).	10.0
navigation_swirl_gain	Weight applied to the swirl behavior in navigate mode.	0.0
navigation_teleopt_gain	Weight applied to the teleoperation behavior in navigate mode.	1.0
occupy_avoid_obstacle_gain	Weight applied to the avoid_obstacle behavior in occupy mode.	1.5
occupy_avoid_robot_gain	Weight applied to the avoid_robot behavior in occupy mode.	2.0
occupy_avoid_robot_sphere	Other robots beyond this range (in meters) are not considered by the avoid robot schema in occupy mode.	30.0
occupy_avoid_robot_min_range	If the robot is within this range of another robot (in meters), the repulsion vector from it is set at a maximum value when in occupy mode.	15.0
swirl_sphere	The maximum distance that obstacles are reacted to by the swirl behavior (in meters).	1.0
swirl_safety_margin	The minimum distance that obstacles must be from the robot to be included in the swirl computation (in meters).	0.1
swirl_open_space	The minimum distance that must be clear in a particular direction before it can be chosen as the heading by the swirl behavior (in meters).	0.1
swirl_open_sphere	The maximum distance (in meters) that obstacles are reacted to for use in determining the heading by the swirl behavior.	0.1

Table 4: Robot definition parameter keyword descriptions

schema-based navigation and associated parameters is given in [2]. In addition, the several keywords can be set to select the type of device to control. See Table 5. The keywords described in Table 6 are

KEYWORD	VALUE DESCRIPTION	Default
robot_type	Selects the type of robot to control. Supported values are: PIONEERAT, URBAN, HOLONOMIC, DRV1, MRV2, MRV3, HUMMER, and UNICYCLE_HUMMER.	HOLONOMIC
run_type	Determines if the control program will drive a simulated robot or actual hardware. (Controlling actual hardware will obviously not work if you don't have any hardware. If you do have actual hardware and want to control it, contact us at Georgia Tech for assistance.) Supported values are: SIMULATION, and REAL.	SIMULATION
robot_color	The robot's color in a simulation.	blue

Table 5: Robot definition parameter device type selection keyword descriptions

only relevant if **run_type** is *REAL* and the hardware is set up appropriately.

For the version running on linux, there is a unfixed memory handling bug, which causes crash if color or hostname parameter is missing. So please specify the full list in those cases.

- **Unit description:** The units to be used in the mission are described in this entry. This entry has several permissible formats. A simple one is:

UNIT <Name> $R_1 R_2 \dots R_n$

where R_i is a robot identifier such as **ROBOT** for a generic robot. This unit can be referred to using the specified name *Name*. (Note the use of angle brackets < and > around the unit name. The angle brackets are a necessary part of the syntax.) Other robot identifiers may be defined with the **NEW-ROBOT** command described above. Units are often more complicated. Therefore, the entry can have a more complex, recursive structure. To specify a unit with two named subunits, use the following syntax:

UNIT <Name> (<Name-1> $R_1 R_2 \dots R_n$) (<Name-2> $R_1 R_2 \dots R_n$)

where R_i can be a robot name, as above, or the embedded definition of another unit. This can be repeated recursively, as necessary.

The main point to understand is that each unit or subunit must have a name if commands are to be given to it.

- **Setting system parameters:** Various system parameters can be set which affect the operation of the system. Each of the possible commands will be listed here with a brief explanation. A number of these commands take an optional boolean argument **ON/OFF**. If neither **ON** nor **OFF** is specified, **ON** is assumed. (Boolean arguments can also use **TRUE/FALSE** or **YES/NO** as equivalents to **ON/OFF**.)

- **SET SEED** *seed*

Set the seed for the random number generator that is used to generate the obstacles randomly. The seed should be an integer number.

- **SET SHOW-TRAILS** [**ON/OFF**]

If set **ON**, the robots leave trails as they move.

- **SET HIGHLIGHT-REPELLING-OBSTACLES** [**ON/OFF**]

If set **ON**, when a robot is close enough to an obstacle for it to be repelled by it, a circle will be drawn around the obstacle outlining its zone of influence. If the robot gets closer than the safety margin, a red circle will be drawn at the diameter of the safety margin.

- **SET SHOW-MOVEMENT-VECTORS** [**ON/OFF**]

If set **ON**, the a red line will be drawn in each motion cycle indicating the direction (and magnitude) of the motion that the robot desires to complete.

⁶The manually-coded-type-robot described in Section 3.2. It does not apply for the *CfgEdit*-type robot.

KEYWORD	VALUE DESCRIPTION	Default
tty_num	Select the serial port to use to talk to the robot. (The value should be an integer such as 0 or 1 and is used to form a device name such as /dev/tty0 or /dev/tty1.)	0
timeout	Select the packet timeout delay in 1/10ths of a second used when talking to the robot. (The value should be an integer such as 10 or 15.)	40 (4 seconds)
use_lawn	Is the serial port connected to the robot using LAWN radio modems? (The value should be either YES or NO.)	NO
lawn_name	If use_lawn is YES, this specifies the name of the LAWN radio mounted on the robot. (The value should be a text string.)	ROBOT
echo_tty	For debugging purposes, the device driver can be made to echo all robot communications to stdout. (The value should be either YES or NO.)	NO
log_tty	For debugging purposes, the device driver can be made to log all robot communications to a file named "tty.log". (The value should be either YES or NO.)	NO
draw_obstacles	Only valid when running a real robot. It causes the robot to report obstacle sensor readings back the simulation where they are drawn as filled circles. (The value should be either YES or NO.)	YES
lurch_mode	Only valid when running a real robot. It causes the robot to halt after each control loop cycle. Normally the controller commands velocities to the robot, allowing the robot to continue on between control loops. When in lurch mode, the robot is commanded to move to a series of discreet destinations. This is a safer mode of execution when there is a chance the control loop might die, since the robot will stop after the end of a move instead of rolling on with nothing in control. (The value should be either YES or NO.)	NO

Table 6: Robot definition parameter keywords for REAL robot execution

- SET DEBUG-ROBOTS [ON/OFF]
If set ON, the robots will print diagnostic information as they run. This information will probably be most useful for the software developers.
- SET DEBUG-SIMULATOR [ON/OFF]
If set ON, the console will print diagnostic information as commands are executed. This information will probably be most useful for the software developers.
- SET DEBUG-SCHEDULER [ON/OFF]
If set ON, the C-Threads scheduler will print diagnostic information as it operates. This information will probably be most useful for the software developers.
- SET ROBOT-LENGTH *length*
This allows the user to set the value of length of the robot. If the robots are scaled (see SET SCALE-ROBOTS below), then this length value is in meters. If not, the value length is the length of the robot in pixels. This command should be placed after the overlay file command since the MISSION-AREA command in the overlay file will override its effect. Note that this only affects the display and does not affect the size of the robot for various computations such as avoiding obstacles.
- SET SCALE-ROBOTS [ON/OFF]
If set ON, the robots are scaled (magnified or shrunk) with the map display.
- SET ZOOM-FACTOR *zoom*

where *zoom* is the desired display zoom factor, in percent. The default is 100%. The minimum value is 10% and the maximum value is 400%. Values beyond that range can be achieved by modifying the values in the `MISSION-AREA` command in the overlay file.

- `SET OBSTACLE-COVERAGE` *number*

This allows the user to set the desired value of the obstacle coverage, in percent.

- `SET MIN-OBSTACLE-RADIUS` *number*

This allows the user to set the desired minimum obstacle radius when obstacles are generated randomly.

- `SET MAX-OBSTACLE-RADIUS` *number*

This allows the user to set the desired maximum obstacle radius when obstacles are generated randomly.

- `SET CYCLE-DURATION` *number*

This allows the user to set the number of seconds each sense-compute-move cycle is supposed to take, in simulation.

- **Create Obstacles:** *The following directive can be used to generate the field of non-overlapping obstacles randomly.*

`CREATE-OBSTACLES`⁷

The following command creates one obstacle at a specified position:

`OBSTACLE` *x y diameter*

*where the position, *x* and *y*, and diameter are all in meters. No checking is done to make sure that these obstacles do not overlap other obstacles or robots.*

- **Print information:** *Several types of information can be printed by including the following commands. These commands are primarily for diagnostic use.*

- `PRINT CONSOLE-DB`

This causes the information in the console database to be printed out.

- `PRINT ROBOT-INFO`

This causes the information about the defined robots to be printed out.

3.6.2 Mission Command List Part

The rest of the command description file is a list of steps (composed of several commands) to be executed. This part of the file starts with the phrase:

`COMMAND LIST:`

This ends the mission background information part of the file. The rest of the file is a series of descriptions of each step. Each step description has the form:

Step-name. UNIT Unit-name CMD additional-information...

⁷The latest version of this function uses fixed values for the minimum and maximum obstacle radius (2m and 10m respectively). Depending on the mission size these values are corrected to be within 1/1000 and 1/5 of the minimum of the width and height of the mission area. You can change these values using `SET MIN-OBSTACLE-RADIUS` and `SET MAX-OBSTACLE-RADIUS`.

You can restore the old functionality by setting the constant `USE_OLD_RANDOM_OBSTACLE_GENERATOR` in `src/mlab/gt-world.c` to a value unequal 0. Note, however, that in this old version the default values depend on the mission area AND the screen resolution.

where *Step-name* is usually a step number (which must end in a period), *Unit-name* is the unit which is to receive this command, *CMD* is a command tag which identifies what command is involved (START, MOVETO, TELEOPERATE, FOLLOW, OCCUPY, or STOP), and the rest of the information on the line varies depending on the type of command. The various command tags (*CMD*) are given in the following sections. The case of the tags is ignored.

A step can be composed of several commands which are to be executed at the same time. To add additional commands to the step, use the token AND between them. For example, a step with two commands would have the following format:

```
Step-name.  UNIT Unit-name CMD additional-information... AND
            UNIT Unit-name CMD additional-information...
```

Additional commands can be appended using further AND tokens.

A description of the format for each type of command⁸ follows:

- **Start command:** This command tells the system to initialize this unit. It has a simple format:

```
UNIT Unit-name START Start-name [dx dy]
```

Where *Start-name* indicates the name of the place where the robots are to be initially located. The optional *dx* and *dy* are offset values for the locations of successive robots in the unit (east and north, respectively); they can be used to keep robots from being placed on top of each other.

- **Moveto command:** This command tells the unit to move to some specified position. It has the following format:

```
UNIT Unit-name MOVETO Destination-name [FORMATION] [FN] [MT] [PLS] [SPEED = number]
```

where *Destination-name* refers to a previously defined destination control measure. If the destination is a line or a region, its centroid is computed and used. The terms in square brackets are optional and can be given in any order. The term *FN* is a formation name which is one of COLUMN, LINE, DIAMOND, or WEDGE. The default formation is no formation. The term *MT* is a movement technique name (TRAVELING, TRAVELING-OVERWATCH, or BOUNDING-OVERWATCH). The default movement technique is TRAVELING. In the current release, traveling overwatch and bounding overwatch are not implemented. The optional speed is in meters per second.

The final part, *PLS*, is an optional phase line specification. It has several possible forms:

```
PHASE-LINE mm-dd-yy hh:mm:ss
```

In this form, the desired arrival date and time are specified. The order of the date and the time can be reversed. The seconds (and the preceding colon) can be omitted from any time specifications. If the date is omitted, the time is assumed to be for the current day:

```
PHASE-LINE hh:mm:ss
```

If a “+” is put in front of a time, it is assumed to be relative to the current time.

```
PHASE-LINE +hh:mm:ss
```

If a number is given with no colon, it assumed to be in minutes:

```
PHASE-LINE +number
```

Although the phase line specification can be given in command lists, currently it is ignored in command execution.

⁸Except “Start command” which can be used for the *CfgEdit*-type robot, these commands only apply for the manually-coded-type-robot described in Section 3.2.

- **Teleoperate command:** This command puts the unit under the control of the teleoperation interface. It has the following format:

UNIT *Unit-name* TELEOPERATE *Destination-name* [FORMATION] [FN] [SPEED = *number*]

The destination-name is currently ignored. The other components are the same as in the MOVETO command. When the user has moved the unit as desired, pressing the [End Teleoperation] button on the teleoperation interfaces causes this command to be completed. One current limitation is that only one unit can be controlled with this command per step.

- **Follow command:** This command tells the unit to follow some line feature. It has the following format:

UNIT *Unit-name* FOLLOW *Destination-name* [FORMATION] [FN] [MT] [it PLS] [SPEED = *number*]

In this case, the destination must be a line. The other components are the same as in the MOVETO command. NOTE: This command is not implemented yet in the robot executable.

- **Occupy command:** This command tells the unit to occupy a position until some termination condition is satisfied. It has the following form:

UNIT *Unit-name* OCCUPY *Destination-name* [FORMATION] [FN] [UC]

Where the *Destination-name* refers to the place to occupy. The formation terms are the same as in the MOVETO command. The final part is the optional termination condition (“Until Clause”). By default, when the robots complete an occupy command (get into formation at the specified location), the console prompts the user to proceed with a proceed dialog box. See Figure 61. When the user



Figure 61: Proceed dialog box

presses the [Proceed] button, the commands continue executing from that point. It may be convenient to specify a timeout for how long to wait for the user to press the [Proceed] button. The form of the “until clause” UC is:

UNTIL TIMEOUT *time*

The value *time* is the amount of time to wait before proceeding without confirmation. If *time* is an integer number, it is assumed to be the number of seconds to wait. It can also be specified in terms of hours and minutes in the format *mm:ss* or *hh:mm:ss*. If a value of zero is specified, the execution proceeds without waiting or putting up a proceed dialog box. The user can also specify a set completion time by using the form:

UNTIL *completion-time*

Where *completion-time* is a time and date in the same format as specified in the phase line for the “Moveto” command. If the date is omitted, the time is assumed to be day the commands are loaded. When that time is reached, the execution proceeds.

- **Stop command:** This command tells the system to stop this unit. It has this form:

UNIT *Unit-name* STOP

This involves killing all the processes related to this unit. Therefore, no commands later in the script should refer to this unit or any units which share robots with this unit (since all these robots are now deactivated)—unless the script restarts the unit with a later START command. This often means that the STOP command is the last command in a script. This also means that the user should use care in

executing commands manually, since using STOP manually while a script is running could produce a situation where the system attempts to execute a command for a robot is dead.

- **Quit command:** The command QUIT can be added to the list of commands. When it is executed, the program stops and exits. This is useful in automated execution of command scripts.
- Most of the SET and PRINT commands described earlier can be embedded in the command lists.

Notice that all the names of control measures in the command descriptions can be preceded by their tag (the *CM* values from Section 3.7) and a dash. This is allowed (and encouraged) to improve readability of the commands. For instance, assembly area Alpha can also be referred to as AA-Alpha in the command list.

3.6.3 Example Command Description File

An example of a command description file is given below. Notice that the file references an overlay file. Also notice the readable nature of the command language:

```
-- THIS IS PART 1: the mission background information
MISSION NAME "Demo C simulation"
SCENARIO "Demo-C"
OVERLAY test.odl
SP Home 0 0
NEW-ROBOT Robot2 robot (base_velocity=5)
UNIT <Wolf> (<Wolf-1> ROBOT ROBOT) (<Wolf-2> ROBOT2 ROBOT2 ROBOT2)

COMMAND LIST:      -- THIS STARTS PART 2: the command list
0. UNIT Wolf START SP-Home 10 0
1. UNIT Wolf MOVETO AA-Alpha FORMATION diamond traveling-overwatch
1a. UNIT Wolf OCCUPY AA-Alpha FORMATION Column
2. UNIT Wolf MOVETO ATK-Bravo FORMATION Column
3. UNIT Wolf OCCUPY ATK-Bravo FORMATION Diamond UNTIL TIMEOUT 20
4. UNIT Wolf MOVETO PP-Charlie FORMATION Column
5. UNIT Wolf FOLLOW Gap-Delta FORMATION Column
6. UNIT Wolf FOLLOW Axis-Foxtrot FORMATION Diamond Traveling-Overwatch
   PHASE-LINE PL-Gamma 06-10-94:23:10
7. UNIT Wolf PASS-PHASE-LINE PL-Gamma
8. UNIT Wolf-1 MOVETO BP-1 FORMATION Line Bounding-Overwatch AND
   UNIT Wolf-2 MOVETO BP-2 FORMATION Line Bounding-Overwatch
9. UNIT Wolf-1 OCCUPY BP-1 FORMATION DIAMOND UNTIL TIMEOUT 10 AND
   UNIT Wolf-2 OCCUPY BP-2 FORMATION DIAMOND UNTIL TIMEOUT 10
10. UNIT Wolf-1 MOVETO OBJ-Zulu FORMATION Line Bounding-Overwatch AND
   UNIT Wolf-2 MOVETO OBJ-Zulu FORMATION Line Bounding-Overwatch
11. UNIT Wolf OCCUPY OBJ-Zulu FORMATION Diamond
12. UNIT Wolf STOP
```

3.7 ODL (Overlay Description Language)

The Overlay Description Language describes the environment for display by *mlab*. In order to read overlay information from files, it is necessary to use an overlay description file format. This section describes the structure and syntax of that format. The end of the section contains an illustration of a sample overlay file in Figure 62.

The file is organized into two parts (1) the scenario information part and (2) the control measure description part. The basic syntax of the file is described in the following sections. White space (spaces, tabs, and blank lines) is ignored. Comments can be included on any line by using two dashes in a row; the rest of the line is ignored.

NOTE: In the following descriptions, items in italics would be replaced by appropriate values from the user.

3.7.1 Scenario Information Part

The scenario information part is the first part of the file and includes any of the following items:

- **Scenario name:** This entry names the scenario for which this overlay data pertains. It has the following format:

```
SCENARIO "scenario name"
```

- **Site name:** This entry indicates the site on which the overlay information is superimposed. This would might eventually include exact information about exactly what map should be accessed. It has the following format:

```
SITE "site name"
```

- **Mission area:** This entry indicates the extent of of the mission area. The format for this command is:

```
MISSION-AREA width height
```

where *width* is the width (east-west) of the mission area and *height* is its height (north-south) — both in meters. This mission area is scaled appropriately to fit within the scrollable map area. The default mission area is 1km by 1km. Note that this command forces recomputation of a number of length-related values and should be including early in the overlay file.

- **origin location:** This entry indicates the origin for the overlay. In other words, the lower left hand corner of the map display area is considered to be at these coordinates. For convenience, all the coordinates in the following control measures are with respect to this origin. The format of the command is:

```
ORIGIN X Y
```

where *X* is value of the east-west origin and *Y* is the value of the north-south origin. In the following control measures, *x* increases in the eastward direction and *y* increases in the northward direction from this origin. If this statement is omitted, an origin of zero is assumed for *x* and *y*.

- **Zoom factor:** This entry allows the user to change the extent to which the map is magnified or shrunk. The format for the command is:

```
SET ZOOM-FACTOR zoom
```

Where *zoom* is the desired zoom factor, in percent. The default is 100%. The minimum value is 10% and the maximum value is 400%. Values beyond that range can be achieved by modifying the values in the MISSION-AREA command. This command should be placed after the MISSION-AREA command, if it is present.

3.7.2 Control Measure Description Part

This part of the file contains a description of all the control measures necessary for this scenario. This part of the file starts with the phrase:

CONTROL MEASURES:

This ends the scenario description part of the file. The rest of the file is a series of descriptions of the various control measures. Most of the descriptions have this general form:

CM Name coordinate-list

where *CM* is a tag specific to each type of the control measure, *Name* is the users unique name for that control measure, and *coordinate-list* is a list of the coordinates necessary to describe the control measure. Note that some of the control measures have variations on this syntax for optional arguments.

If the user does not want the name for the control measure to be printed on the display, they may append an asterisk at the end of the name. For instance, consider these two control measures:

CM George coordinate-list *CM Bill* coordinate-list*

Control measure for George would have its name printed on the display, whereas the control measure for Bill would not. Bill's graphics would still print, but the identifying name would not. This is useful in some situations to reduce the clutter on the screen. However, please name each control measure with a unique name. Currently, if there are redundant names for different control measures, some of them would not be drawn correctly.

The various tags (*CM*) are given below. The case of the tags is irrelevant. Each user-supplied name (*Name*) must be unique. If the user wishes to supply a number for a name, it must be done in double quotes (eg, "1"). Descriptions of each type of control measure follow. All lengths and distances are in meters.

- **Assembly Area:** This entry indicates an assembly area. It has the format:

AA *Name* $x_1 y_1 \quad x_2 y_2 \quad \dots \quad x_n y_n$ [*diameter*]

where the coordinates describe the polygon defining the region involved. The last point is assumed to be connected to the first. The optional value *diameter* (in meters, default=50m) can be specified if only the coordinate of the center of the position is given—then the position is assumed to be circular with the specified diameter. (If multiple coordinates are given for the border of the position, any specified diameter will be ignored.)

- **Attack position:** This entry indicates an attack position. It has the format:

ATK *Name* $x_1 y_1 \quad x_2 y_2 \quad \dots \quad x_n y_n$ [*diameter*]

where the coordinates describe the polygon defining the region involved. The last point is assumed to be connected to the first. The optional value *diameter* (in meters, default=50m) can be specified if only the coordinate of the center of the position is given—then the position is assumed to be circular with the specified diameter. (If multiple coordinates are given for the border of the position, any specified diameter will be ignored.)

- **Axis:** This entry indicates an axis of advance. It has the format:

Axis *Name* $x_1 y_1 \quad x_2 y_2 \quad \dots \quad x_n y_n$ [*width*]

where the coordinates describe the center line of the axis. The optional final value is the width of the axis in meters; it defaults to 100m if unspecified.

- **Battle position:** This entry indicates a battle position. It has the format:

BP *Name* $x_1 y_1 \quad x_2 y_2 \quad \dots \quad x_n y_n$ [*diameter*]

where the coordinates describe the polygon defining the region involved. The last point is assumed to be connected to the first. The optional value *diameter* (in meters, default=50m) can be specified if only the coordinate of the center of the position is given—then the position is assumed to be circular with the specified diameter. (If multiple coordinates are given for the border of the position, any specified diameter will be ignored.)

- **Boundary:** This entry indicates one of the boundaries in the scenario. It has the format:

Boundary *Name* $x_1 y_1 \quad x_2 y_2 \quad \dots \quad x_n y_n$

where the coordinates describe the line of the boundary. Each boundary entry describes one boundary. It is likely that several entries will be necessary since many scenarios have several boundaries.

- **Door:** This entry indicates a door object of the overlay. It has the format:

Door *Name* $x_1 y_1 \quad x_2 y_2$

where the coordinates describe the line associated with a door. The robot can detect this control measure as a “door object”, and can be marked or unmarked.

- **Gap:** This entry indicates a gap. It has the format:

Gap *Name* $x_1 y_1 \quad x_2 y_2$ [*width*]

where the coordinates describe a straight line segment from one end of the gap to the other. The final, optional term *width*, is the width of the gap in meters; it defaults to 40m if unspecified.

- **Hallway:** This entry indicates a hallway object of the overlay. It has the format:

Hallway *Name* $x_1 y_1 \quad x_2 y_2 \quad x_3 y_3 \quad x_4 y_4 \quad \dots \quad x_n y_n$

where the first two points describe the center line of the hallway, and the rest of the points describe a closed polygon. When the robot is inside this polygon, it can be distinguished as it is in the hallway.

- **Line of Departure/Line of Contact:** This entry indicates a line of departure/line of contact. It has the format:

LDLC *Name* $x_1 y_1 \quad x_2 y_2 \quad \dots \quad x_n y_n$

Note that the friendly forces are assumed to be on the right side of this line (heading from its beginning towards its end).

- **Objective:** This entry indicates an objective. It has the format:

OBJ *Name* $x_1 y_1 \quad x_2 y_2 \quad \dots \quad x_n y_n$ [*diameter*]

where the coordinates describe the polygon defining the region involved. The last point is assumed to be connected to the first. The optional value *diameter* (in meters, default=50m) can be specified if only the coordinate of the center of the objective is given—then the objective is assumed to be circular with the specified diameter. (If multiple coordinates are given for the border of the objective, any specified diameter will be ignored.)

- **Object:** This entry indicates a circular object. It has the format:

OBJECT $x y$ *diameter* [Fixed — Movable — Container] “*color*”

where x and y specify the center of the obstacle and *diameter* gives its diameter, all in meters. The three classes of objects are as follows:

- Fixed objects can not be moved and are usually obstacles
- Movable objects can be manipulated by the robots
- Container objects can hold movable objects placed there by the robots. Objects in a container are not visible to robots.

- **Passage Point:** This entry indicates a passage point. It has the format:

PP *Name* *x* *y* [*diameter*]

where *x* and *y* describe the location of the passage point. The optional value *diameter* (in meters, default=10m) gives the diameter of the passage point.

- **Phase line:** This entry indicates a phase line. It has the format:

PL *Name* *x*₁ *y*₁ *x*₂ *y*₂ ... *x*_{*n*} *y*_{*n*}

- **Room:** This entry indicates a room object of the overlay. It has the format:

Room *Name* *x*₁ *y*₁ *x*₂ *y*₂ ... *x*_{*n*} *y*_{*n*}

where the coordinates describe the closed polygon. When the robot is inside this polygon, it can be distinguished as it is in the room.

- **Starting point:** This entry describes a starting location. It has the format:

SP *Name* *x* *y*

- **Wall:** This entry indicates a wall object of the overlay. It has the format:

Wall *Name* *x*₁ *y*₁ *x*₂ *y*₂ ... *x*_{*n*} *y*_{*n*}

where the coordinates describe the line of the wall. Each boundary entry describes one wall. As in Boundary, it is likely that several entries will be necessary since many scenarios have several walls. This control measure is implemented with the obstacle functionality. In other words, the robots will avoid the walls as if they are obstacles. Moreover, when the robot is in the detect-object mode, any object which is located beyond this wall cannot be detected, since the robot cannot see through the wall.

- **Waypoint file:** This entry indicates the name of the waypoint file to be loaded. The format is:

WAYPOINT-FILE *Name*

This line is usually added by *mlab* when the user chooses to save the overlay file after he or she creates waypoints. The extension of the waypoint file is “.wpt”, and it contains the coordinates of waypoints described as the passage points (PP). Since the overlay parser expects this entry to be the last entry of control measures, the user cannot write below this line.

3.7.3 Example Overlay Description File

To provide a more concrete feel for what an overlay description file might look like, a sample file is shown in Figure 62. The file does not have the numbers specified.

```
-- THIS IS PART 1: The scenario information

SCENARIO "Demo-C"

SITE "Demo B site, Colorado"

ORIGIN X Y

CONTROL MEASURES:  -- THIS STARTS PART 2: The control measure descriptions

Boundary Yankee x1 y1 x2 y2 ... xn yn

LDLC Echo x1 y1 x2 y2 ... xn yn

AA Alpha x1 y1 x2 y2 x3 y3 ... xn yn

ATK Bravo x1 y1 x2 y2 x3 y3 ... xn yn

PP Charlie x y

Gap Delta x1 y1 x2 y2

Axis Foxtrot x1 y1 x2 y2 ... xn yn

PL Gamma x1 y1 x2 y2 ... xn yn

BP 1 x1 y1 x2 y2 ... xn yn

BP 2 x1 y1 x2 y2 ... xn yn

OBJ Zulu x1 y1 x2 y2 ... xn yn

OBSTACLE x1 y1 10.0
```

Figure 62: Sample Overlay Description File.

(Obviously the x and y values would be replaced by actual numbers.)

3.8 *HServer* (Hardware Server)

HServer provides a control interface to all the robot hardware, and you can monitor and configure the hardware through this interface. If you create a robot mission using *CfgEdit* (Section 3.3), and wish to run it on a real robot, the robot executable created using *CfgEdit* will serve as *HClient* and talk to *HServer* using IPT (Section 4.1). *HServer* contains all the hardware drivers, and can control the robot hardware exactly as the robot executable requested. Thus, the robot executable does not even need to know what type of robots it is running on. Since *HClient* and *HServer* communicate each other through IPT, the robot executable is not required to run on the same machine as *HServer*. By separating the hardware drivers from the robot executable binary, *CfgEdit* can compile the mission with less time, and since *HServer* is multithreaded (pthreads), it provides the ability to block on serial reads. If the hardware control was in the robot executable, no blocking I/O could be utilized. Moreover for developers, since *HServer* provides a standard framework for all the hardware drivers, it minimizes the work needed to integrate new hardware into the existing system. *HServer* can control the following hardware:

- ATRV-Jr robot (iRobot) via TCP/IP (*mobility*⁹ required).
- Urban Robot (iRobot) via TCP/IP (*mobility* required).
- AmigoBot robot (ActivMedia) via a serial port.
- Pioneer AT robot (ActivMedia) via a serial port.
- Pioneer 2DX robot (ActivMedia) via a serial port.
- EVI-D30 camera (Sony) via a serial port.
- LMS200 Laser scanner (SICK Optic Electronic) via a serial port.
- Nomad 150/200 robot (Nomadic Technologies, Inc.) via a serial port or TCP/IP.
- Cognachrome Vision System (Newton Research Labs.) via a serial port.
- BT848 Video Framegrabber via /dev/video/video0.
- Novatel GPS system via a serial port.
- GPS base station via a serial port.
- DMU-VGX Gyroscope (Crossbow) via TCP/IP (*mobility* required).
- C100 Compass (KVH) via TCP/IP (*mobility* required).
- X server via local or TCP/IP.

3.8.1 Configuration File

HServer requires a configuration file to run. The purpose of the configuration file is to identify the hardware settings and connectivity. The configuration file is broken down into sections, each of which specifies possible connectivity/settings for a particular piece of hardware such as a robot or peripheral device such as a laser. The default file is “.hserverrc”, although an alternate file may be specified in the command line. To find the configuration file, the following places will be searched in the following order: the current directory, the user’s home directory, and the directories in the user’s PATH environment variable. When MissionLab is installed, the default .hserverrc file can be found in [your MissionLab home]/src/hardware_drivers/hserver directory.

⁹*HServer* has to be compiled with iRobot’s *mobility* by “make hserver_mobility”. Because of the copyrights, however, *mobility* is not included in the *MissionLab* distribution. See <http://www.irobot.com/rwi> for how to obtain *mobility*.

The ‘#’ character denotes a comment. Any text following the ‘#’ character up to the end of the line is ignored.

Other than comments, the file is composed of sections with the following format:

```
<start [section type] [section name]>
[value name 1] = [value 1]
[value name 2] = [value 2]
[value name 3] = [value 3]
...
[value name n] = [value n]
<end>
```

Each section has a name, [section name], which uniquely identifies that type of section. All names of sections of the same type must be unique, but if two sections are of different types, they may have the same name. Names may not have spaces. If a value name/value pair is left out, a default value is used. Usually this is the empty string for strings and 0 for numbers, but there are exceptions (covered later).

Value names represent settable options pertaining to the type of hardware the section represents. For example, *laser* sections have a *port_string* value. This value defines the serial port (such as */dev/ttyS0*) *hserver* will use to try to connect with the laser. Value names are not case sensitive.

Below is an example of a section for a device of type *laser* with the name *laserS1*. The specialized value names for this and other devices will be described in detail later on.

```
<start laser laserS1>
name = front
port_string = /dev/ttyS1
angle_offset = 0
x_offset = 0 # in cm
y_offset = 0 # in cm
listen_ipt = false
send_ipt = false
#stream_host = localhost # used for streaming
<end>
```

Values are strings that may contain spaces. Values start with the first non-white space character after the ‘=’ character which follows the value name and continues to the end of the line or the ‘#’ character. Trailing white space is deleted. For example, the lines

```
port_string=/dev/ttyS0
port_string= /dev/ttyS0
port_string = /dev/ttyS0# this is the port string
port_string=/dev/ttyS0 # this is the port string
```

will all reduce to having a value of “/dev/ttyS0”. Values may or may not be case sensitive, depending on the context. For example, the boolean value “true”, the following may all be accepted: “true”, “TRUE”, “True”, or “1”. Other values may be more strict, however, such as values that specify port strings. Since IO ports are treated as files, case matters.

Inclusion of a section in the configuration file does not mean that *hserver* will connect to that piece of hardware. Instead, that section may now be specified in the command line. Actual hardware connections are executed only if a configuration section is specified in the command line (see the section on command line arguments below) or by interactively telling *hserver* to connect to a piece of hardware while it is running.

Currently supported section types and their options are:

- camera

This represents options for a Sony evi-D30. Currently defined values are:

- port_string - This defines what serial port hserver should use to connect to the camera. Default is the empty string.

- laser

This represents options for a SICK LMS200 laser scanner. Currently defined values are:

- name - This is an identifier for this laser (not the same as the section name). No two lasers running at the same time should have the same name. Default is the empty string.
- port_string - This defines what serial port hserver should use to connect to the laser. Default is the empty string.
- angle_offset - The direction the laser is pointing, in degrees. The front of the robot is zero degrees. Default is zero.
- x_offset - The laser's distance from the center of the robot along the robot's X-axis, where zero is the center of the robot, and positive X is toward the front of the robot. The value is assumed to be in cm. Default is zero.
- y_offset - The laser's distance from the center of the robot along the robot's Y-axis, where zero is the center of the robot, and positive Y is toward the robot's left side. The value is assumed to be in cm. Default is zero.
- listen_ipt - A host name from which to listen for laser readings via ipt. Default is the empty string.
- send_ipt - A host name to which laser readings should be sent via ipt. Default is the empty string.
- stream_host - A host name to which laser readings should be sent via a tcp socket. Default is the empty string.

- cognachrome

This represents options for a cognachrome vision system. Currently defined values are:

- port_string - This defines what serial port hserver should use to connect to the cognachrome. Default is the empty string.

- nomad

This represents options for a Nomad 150 or Nomad 200. Currently defined values are:

- type - This defines the type of nomad to connect to. Valid values are "150" and "200". Default is "150".
- port_string - This defines what serial port hserver should use to connect to the Nomad 150. Default is the empty string. This value isn't needed for Nomad 200s
- host_name - The host name of the Nomad 200. Default is the empty string. This value isn't needed for Nomad 150s.

- amigobot

This represents options for a AmigoBot. Currently defined values are:

- port_string - This defines what serial port hserver should use to connect to the Amigobot. Default is the empty string.

- pioneer

This represents options for a Pioneer AT. Currently defined values are:

- port_string - This defines what serial port hserver should use to connect to the Pioneer AT. Default is the empty string.
- pioneerd
 - This represents options for a Pioneer 2DX. Currently defined values are:
 - port_string - This defines what serial port hserver should use to connect to the Pioneer 2DX. Default is the empty string.
- framgrabber
 - This represents options for a framgrabber. Currently defined values are:
 - port_string - This defines what serial port hserver should use to connect to the framgrabber. Default is the empty string.
- gps
 - This represents options for a GPS unit. Currently defined values are:
 - port_string - This defines what serial port hserver should use to connect to the GPS unit. Default is the empty string.
 - use_base - This is a boolean value that tells whether or not hserver should connect to a GPS base station. Defaults to "false".
 - base_lat - The latitude, in degrees, of the base station. Default is 33.78135 (base station on Ga Tech campus).
 - base_long - The longitude, in degrees, of the base station. Default is -84.40034 (base station on Ga Tech campus).
 - x_diff - The difference, in meters, of the X position as defined by the GPS unit and as defined by the controlling application's coordinate system. Default is 220.
 - y_diff - The difference, in meters, of the Y position as defined by the GPS unit and as defined by the controlling application's coordinate system. Default is 270.
 - m_per_lat - The number of meters per degree latitude. Default is 110919.337 (correct for Ga Tech campus).
 - m_per_long - The number of meters per degree longitude. Default is 92621.190 (correct for Ga Tech campus).

3.8.2 Command Line Arguments

The followings are the command line arguments *HServer* can take. If you run *HServer* with no arguments, it starts up with no hardware connections. Hardware can be then connected via keyboard commands.

- C *config_file* Use this configuration file.
- c *camera_section* Connect Sony EVI-D30 camera described in the configuration file by camera section <*camera_section*>.
- l *laser_section* Connect SICK LMS200 laser scanner described in the configuration file by laser section <*laser_section*>
- m *cognachrome_section* Connect Newton Cognachrome Vision System described by the configuration file in cognachrome section <*cognachrome_section*>
- n *nomad_section* Connect Nomad 150 or Nomad 200 described in the configuration file by nomad section <*nomad_section*>

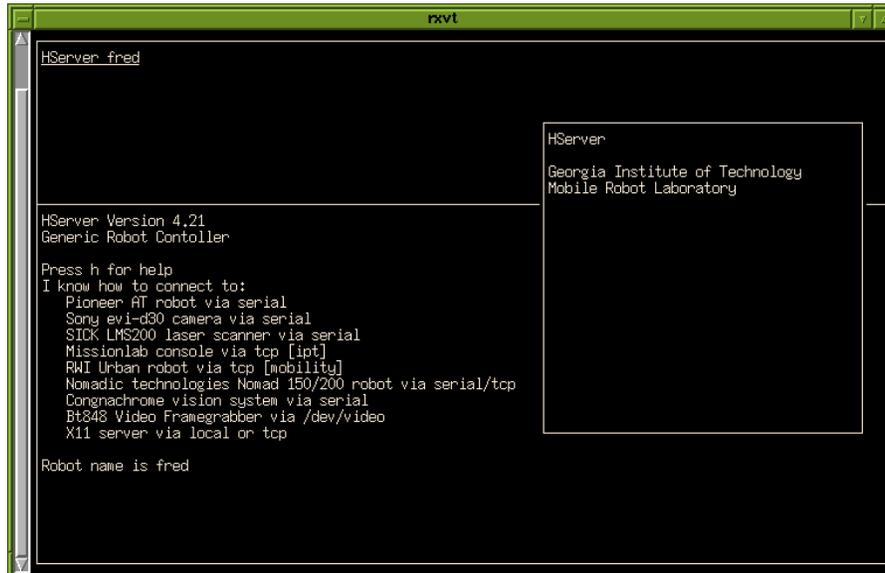
- p *pioneer_section* Connect to a Pioneer AT described in the configuration file by pioneer section <*pioneer_section*>
- d *pioneer_section* Connect to a Pioneer 2DX described in the configuration file by pioneer section <*pioneerd_section*>
- j Connect to an ATRV-Jr via *mobility*
- u Connect to an RWI Urban Robot via *mobility*
- v *framegrabber_section* Connect BT848 Video Framegrabber described in the configuration file by framegrabber section <*framegrabber_section*>
- g *gps_section* Connect to the GPS unit described in the configuration file by gps section <*gps_section*>
- y Connect to a Crossbow DMU VGX gyroscope via *mobility*
- k Connect to a KVH C100 compass via *mobility*
- r *HServer* IPT name defaults to “fred”
- b Display battery information
- f Connect to an emulated robot “fred”
- i [*hostname*] Start IPT. Set ipthost to <*hostname*> (optional)
- a Connect to *iptserver* and listen for the connection with the *HClient* of the robot executable
- o Setup IPT handler for multiple hservers
- x Open X window
- S no splash screen
- h print usage and quit

Note: See the paragraph on “Emulated Robot Control Menu” below for more information on “fred”.

3.8.3 Environment Variables

For your convenience, you can set the following variables in your environment:

- HSERVER_USE_MLAB** If this variable is set in your environment, *HServer* will default to listening for *HClient* of the robot executable. It is same as starting *HServer* with “-a” option
- IPTHOST** Hostname for IPT. If this variable is not set, it defaults to local host.

Figure 63: *HServer* - main window

3.8.4 User Interface and Key Commands

- **Main Window:** The main window is divided into two sections (Figure 63). The top section are the status bars for the requested hardware. These status bars indicate the current state of the hardware and other information. The lower section of the main window is for general output. All error and debugging information will be displayed here. The main window accepts commands to start new hardware and to bring up control menus for connected hardware. In addition to the main window, the figure below also shows the message window located at the center right in the figure. The message window is described below.

The following key commands are accepted by the main window.

- B Display laptop battery information if it is available
- C Start Sony EVI-D30 camera if not connected or show the camera control menu
- c Clear text window
- D Change report levels
- E Display the sensors window
- F Start emulated robot or show the emulated robot control menu
- G Connect to the GPS unit
- h Toggle the showing of the help window
- I Connect to IPT
- K Connect to the KVH C100 compass
- L Start SICK laser scanner or show the laser control menu
- M Start Newton Cognachrome Vision System or show the Cognachrome control menu
- p Pause the text on the screen
- Q Disconnect all hardware and exit, or close the message window if it is open
- R Show the remote control; used to connect to robots or control them
- r Refresh screen

- T Start terminal if it is not connected or show the terminal window
- V Start video or show video control window
- W Show the robot configuration
- X Create xwindow or show xwindow
- Y Connect to a Crossbow DMU VGX gyroscope

- **Help Window:** This window gives a listing of the keyboard commands accepted by the main window. This window will pop-up or pop-down by pressing “h” key.
- **Message Window:** This window is a multi-purpose window which pops up at the center of the main window, displaying “HServer Georgia Institute of Technology ...”, initially. It displays prompts for input and also hardware control menus for individual hardware. This window can be moved by using the arrow keys or the numeric keypad (with numlock on).
- **Terminal Window:** This window, which pops up by pressing “T” key, is for direct connection to a serial device. It takes input from the keyboard and outputs it to the serial port and writes data received from the serial port to the window. Terminal window commands are initiated with the escape key. If you wish to send an escape character¹⁰ (0x1B) over the serial line press the escape key twice. The followings are key commands that supported by the window:

- d Disconnect terminal and close window
- e Erase terminal window
- h Toggle hex mode - all data from the serial port is printed out in hexadecimal
- C Toggle cognachrome mode - only outputs data received on the current watched stream
- c Change cog stream
- p Send ”q” as saphira packet - quits saphira mode on cognachrome
- r Refresh screen
- l Toggles log mode - Terminal output is logged to hserver.log file
- x Hide the terminal window

- **Remote Control Window:** This window provides capability of driving the robot or moving the camera manually. Please note that when this window is up, the commands from *HClient* (*MissionLab* robot executable) do not get received. Thus, make sure to pop down this window when you are running a mission created from *CfgEdit*. This window will pop up by pressing “F” key from the main window. The followings are the key commands you can use to control the robot and/or camera:

- i Move the robot forward
- j Move the robot to the left
- l Move the robot to the right
- k Move the robot backward
- n Lift the left arm of the robot (Urban Robot)
- m Lift the right arm of the robot (Urban Robot)
- p Increase the speed of the robot
- ; Decrease the speed of the robot
- <space> Stop the robot movement
- u Stop the linear motion of the robot, but continue on the angular motion
- o Stop the angular motion of the robot, but continue on the linear motion

¹⁰Note that because *HServer* uses ncurses there is an unavoidable delay associated with the escape character.

- , Stop the movement of the robot arm (Urban Robot)
- w Tilt the camera upward
- a Pan the camera to the left
- d Pan the camera to the right
- s Tile the camera downward
- 0 Set the camera angle to be its center position
- q Zoom in the camera view
- z Zoom out the camera view
- e Increase the speed of the camera movement
- c Decrease the speed of the camera movement
- t Turn on/off the teleport mode - when *HServer* is in the teleport mode, “i”, “j”, “l”, and “k” keys will relocate the robot position, and “u”, and “o” will reset its heading
- x Exit control menu

- **AmigoBot / Pioneer-AT Control Menu:** This window handles the miscellaneous controls of AmigoBot or Pioneer AT robot. If *Hserver* is already connected to the robot, this window will pop up by pressing “R” key from the main window. If the robot is not already connected, then first press “R” to get a menu to connect the robot.

- d Disconnect the robot
- s Synch
- ab Sonar Type (Supported by Pioneer AT/2DX only)
- p Turn on/off sonar sensors
- = Speed Factor Decrement/Increment
- ._+ Angular Speed Factor Decrement/Increment
- r Refresh screen
- x Exit control menu

- **Nomad Control Menu:** This window handles the miscellaneous controls of Nomad 150/200 robot. If *Hserver* is already connected to the Nomad, this window will pop up by pressing “R” key from the main window. If the Nomad is not already connected, then first press “R” to get a menu to connect the robot.

- d Disconnect Nomad
- s Range Start
- + - Change sonar sensor fire rate
- z Zero robot
- c Calibrate Compass
- p Ping
- r Refresh screen
- x Exit control menu

- **ATRV-Jr / Urban Robot Control Menu:** This window handles the miscellaneous controls of ATRV-Jr or Urban Robot. If *Hserver* is already connected to the robot, this window will pop up by pressing “R” key from the main window. If the robot is not already connected, then first press “R” to get a menu to connect the robot.

- d Disconnect the robot
- r Refresh screen
- x Exit control menu

- **Sensors Window:** This window gives a listing of the robot sensors and current sensor information. It is updated twice a second. This window will pop up by pressing “E” key from the main window.
- **Camera Control Menu:** This window handles the miscellaneous controls of Sony camera. This window will pop up by pressing “C” key from the main window.

- d Disconnect camera
- t Set tracking mode - supporting the following modes:
 - none: No tracking
 - center: Center camera
 - sweep X: Sweep pan back and forth
 - sweep Y: Sweep tilt back and forth
 - sweep XY: Sweep from (min pan,0 tilt) \Rightarrow (max pan,0 tilt) \Rightarrow (max pan,max tilt) \Rightarrow (min pan,max tilt)
 - largest object [cognachrome]: Track largest object on all three channels
 - largest A [cognachrome]: Track largest object on channel A
 - largest B [cognachrome]: Track largest object on channel B
 - largest C [cognachrome]: Track largest object on channel C
 - closest laser reading: Track pan to face closest laser reading
- r Refresh screen
- c Center camera
- p Toggle camera power
- x Exit control menu

- **Laser Control Menu:** This window handles the controls of SICK laser scanner. To get to this window, first press “L” from the main window. If there are any lasers to control, you will be given a choice to add or control a laser. (If there are no lasers connected, *Hserver* assumes you want to add a laser, and it begins asking for the information needed to do so.) Choose “2” to control a laser. *Hserver* will present you with a list of connected lasers from which to choose. Once you choose a laser, you will be presented with the following menu.

- d Disconnect laser
- s Stream laser readings over TCP/IP
- c start/stop continual laser updates
- u update laser readings
- r Refresh screen
- x Exit control menu

- **Cognachrome Control Menu:** This window handles the controls of Newton Cognachrome Vision System. This window will pop up by pressing “M” key from the main window.

- 0-2 Set channel to train and view
 - t Train on center color
 - i Incremental train center
- < > Shrink/grow colors

- S R Save/Restore EPROM
- c Toggle crosshairs
- + Adjust filter level
 - Level 0: recognizes objects one pixel wide
 - Level 1: 2 pixels
 - Level 2: 3 pixels
 - Level 3: 4 pixels
- d Disconnect cognachrome
- z Toggle color tracking
- [] Adjust object size used to calculate distance
- x Exit control menu

- **Video Control Menu:** This window handles the controls of BT848 video framegrabber. This window will pop up by pressing “V” key from the main window.

- d Disconnect video
- 0-2 Set channel to digitize
- !@#\$ Set capture size to:
 - !: 160x120
 - @: 320x240
 - #: 480x360
 - \$: 640x480
- c Capture frame - if the xwindow is open output to xwindow (see below), otherwise write to “video_cap.ppm”
- r Refresh screen
- x Exit control menu

- **Xwindow**

This window is for displaying images captured by the video framegrabber. This window will pop up by pressing “X” key from the main window.

- Q Closes xwindow

- **Emulated Robot Control Menu:** The emulated robot (Fred) emulates the movement of an actual robot without actually running the hardware of those. Event though it is similar to the *MissionLab* simulation capability, the movement of the emulated robot is based on the kinematics of the Pioneer AT and Urban Robot hardware while the *MissionLab* simulation assumes that these robots are holonomic. This window can pop up by pressing “F” from the main window.

- d Disconnect the emulated robot
- i Initializes the emulated robot location to be to <5, 5>, and its heading to be 0.

3.9 CBRServer and CBR Mission-Planning Wizard (“Mission Expert”)

3.9.1 Overview

CBRServer (Case-Based Reasoning Server) is part of the CBR mission-planning wizard (we refer it here as “Mission Expert”) that assists users to create a new mission plan based on past successful mission plans stored in the memory. As shown in Figure 64, *CBRServer* serves as a mission-plan database and a mission-planning engine by running as a separate process. The specification of the mission entered by a user (using *mlab*) will be sent to *CBRServer* via *CfgEdit* (using TCP/IP), *CBRServer* returns appropriate mission plans to the *CfgEdit* after assembling them from the segments of the mission plans (cases) stored in the memory, using case-based reasoning.

The relationship between Mission Expert and *CBRServer* is shown in Figure 65. As shown in the figure, interactions among *mlab*, *CfgEdit*, and *CBRServer* processes provide the functionality of Mission Expert. The user would only interact with *mlab* and *CfgEdit*. *CBRServer* consists of Memory Manager, Planner, and Domain Manager modules. To generate a new mission, Memory Manager module retrieves ballpark solutions based on the mission specs (preferences and constraints) provided by the user. Planner module combines those solutions and adjust parameters to make it better fit to the current situation. Domain Manager module translates the solution into a language that *CfgEdit* can understand. Multiple solutions may be suggested to the users if applicable. Each solution will be graded with its suitability rating or “matchness of fit”. Once *CfgEdit* loads the solution (FSA-based mission plan), the user may modify it if necessary. If the user wishes, Mission Expert can save a new mission plan into the memory (See Section 3.9.4 below). After the execution of the mission, Mission Expert inquire the user feedback, and it will be used to adjust the rating of the mission plan in the next cycle.

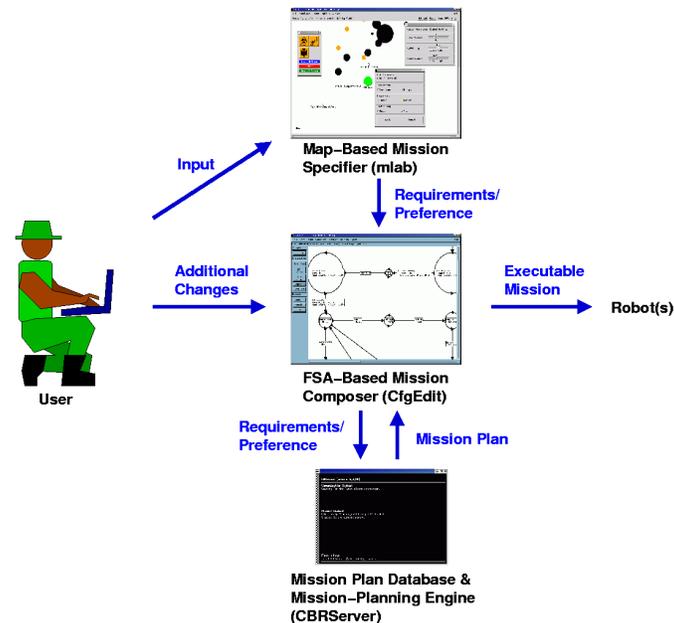


Figure 64: Mission Expert processes: *mlab*, *CfgEdit*, and *CBRServer*.

3.9.2 Invoking *CBRServer*

CBRServer takes the following command line arguments:

```
cbrserver [-dh] [-s socket name] [-l library name]
```

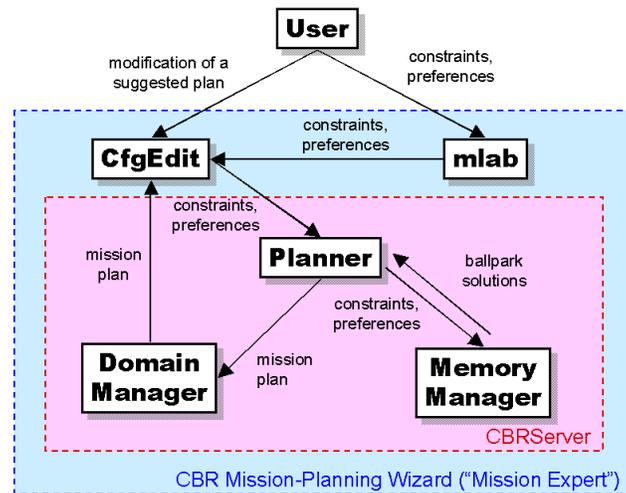


Figure 65: Interactions among the user, *mlab*, *CfgEdit*, and internal modules of *CBRServer*.

-d enables the debugging mode, and it will dump debugging information in a file (whose name would be something like "cbrserver-debug-xxxx-yyyy.out" where xxxx is your user name and yyyy is a current date). -h will print out the usage. -s will set *CBRServer* to use *socket name* to communicate with *CfgEdit*. The default *socket name* is "/tmp/xxxx-cbrplanner.socket" where xxxx is your user name. The default *socket name* can be altered by uncommenting the line for *CBRServerSocketName* in ".cfgeditrc". -l option will make *CBRServer* to load the specified CBR library file ("*.cbl") upon start-up. The invoked *CBRServer* will display the (ncurses) window as in Figure 66.

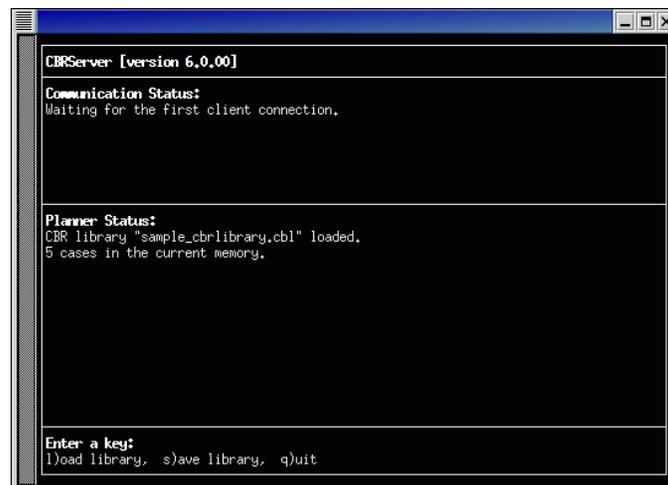


Figure 66: *CBRServer* - main window

The list of the key entries supported by *CBRServer* is shown in the bottom part of the window. To load a CBR library file, enter 1 key. To save the cases (sent by *CfgEdit*) in the memory, enter s key. Enter q key to quit *CBRServer*.

3.9.3 Creating an Example Mission Using Mission Expert

This example shows you how to construct a mission plan using Mission Expert. In this example, you will be making a mission in which three robots will conduct an EOD (explosive ordnance disposal) task after following designated waypoints.

1. Check `.cfgeditrc` Option

- View `[your MissionLab home]/src/cfgedit/.cfgeditrc`.
- Make sure `DisableMissionExpert` is set as “false”.

2. Run `iptserver`

- You must be running `iptserver` when the `CfgEdit` invokes `mlab` (Read Section 3.1 for the explanation). In a separate terminal window, run `iptserver`.

```
% iptserver
```

3. Run `CBRServer`

- Open up another terminal window.
- Go to the demo directory.

```
% cd [your MissionLab home]/demos/mars_demos
```

- Make sure there is an example CBR library file “`sample_cbrplanner.cbl`”.
- Run `CBRServer` with “`-l`” option to load the CBR library file.

```
% cbrserver -l sample_cbrplanner.cbl
```

4. Run `CfgEdit` with Mission Expert

- Open up another (3rd) terminal window.
- Go to the same “`mars_demos`” directory.
- Run `CfgEdit` with the specified configuration file. (Note: the `CBR Wizard` should load the `.mission_expertrc` found in the current directory, not the one from `[your MissionLab home]/bin`.)

```
% cfgedit -c config-cbr
```

- Choose “New Robot” in the first dialog box.
- The second dialog should ask you whether to use Mission Expert (Figure 67). Select “Yes” to the question.

5. Designing a Mission

- At this point, `mlab` should be asking you to select an overlay (Figure 68). Choose “`minefield.ovl`”.
- `mlab` should now popup the windows for designing a mission. Click on “Waypoints Task” in the toolbox to highlight the button (Figure 69).
- Click on anywhere in the field with the left mouse button. It should create an icon for “Waypoints Task”. Repeat once more to create two waypoints (Figure 70).
- Now, click on the “EOD Task” button in the toolbox window.

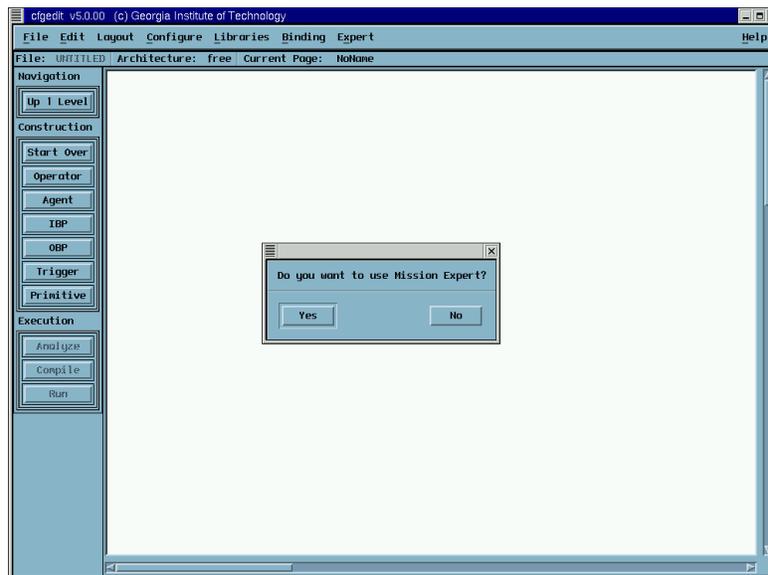


Figure 67: *CfgEdit* with the dialog box asking for Mission Expert usage

- Click on anywhere in the field with the left mouse button again. It should create an icon for “EOD Task” this time.
- View the setup for the EOD task by clicking the “EOD Task” icon with the right mouse button, and change the “Environment” to be “Outdoor” (Figure 71). Click on the “Apply” button to close the window.
- Slide “NumberOfRobots” in the Mission Preference window to use 3 robots (Figure 72).
- Click on the red “Finish” button in the toolbox. It will close the *mlab* session.

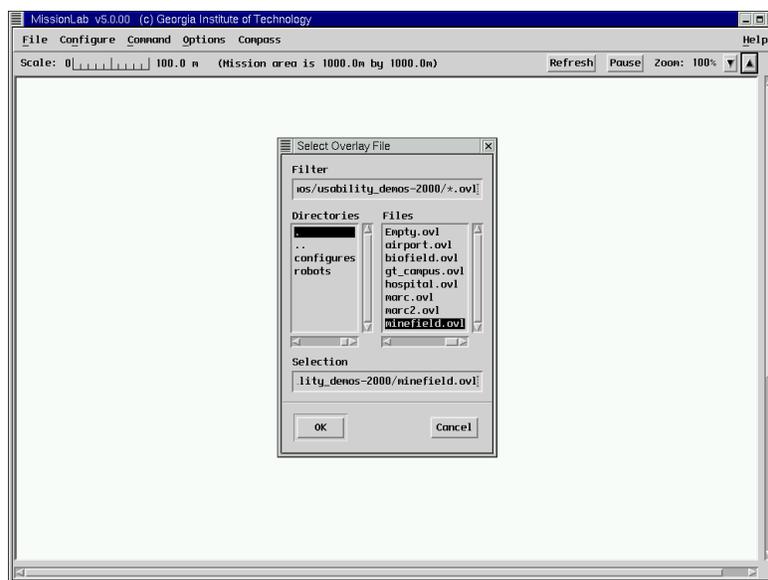


Figure 68: *mlab* asking for an overlay

6. Viewing and Compiling the Generated Mission



Figure 69: Toolbox for selecting desired a task.

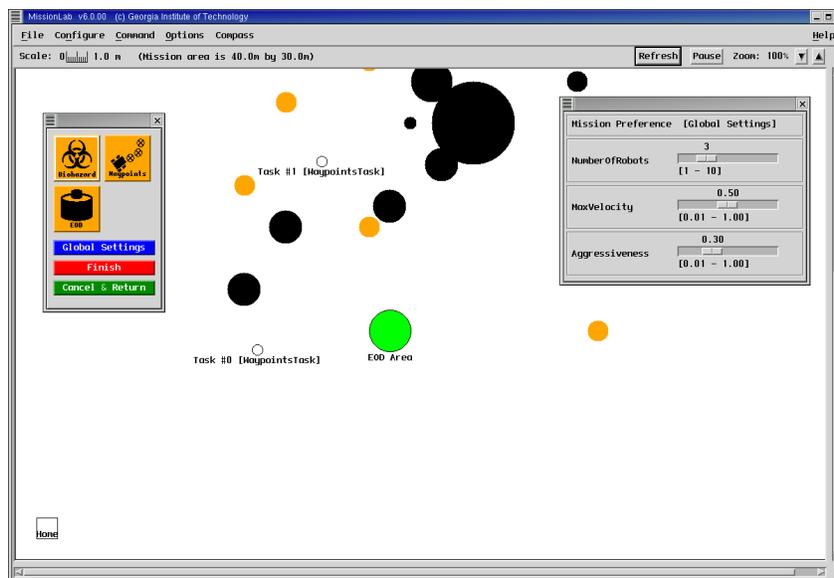


Figure 70: *mlab* showing two waypoints being placed.

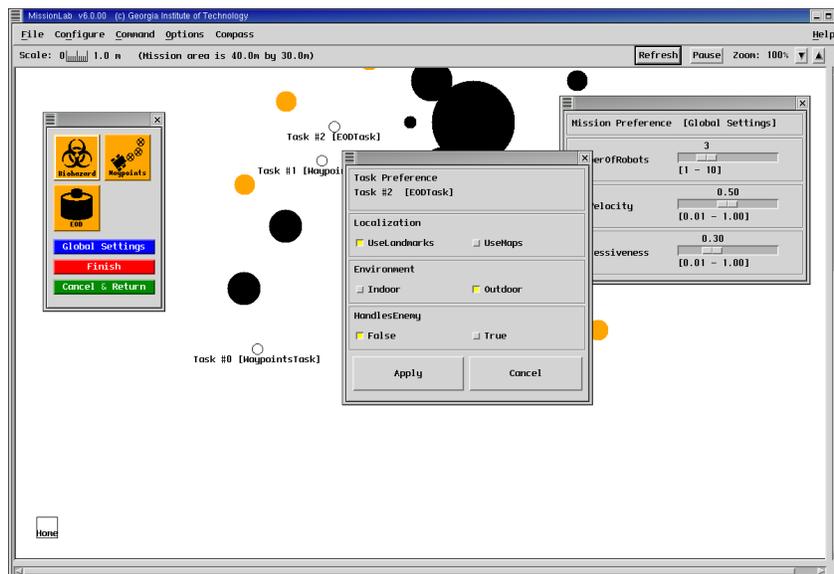


Figure 71: *mlab* showing the setup for the EOD task.

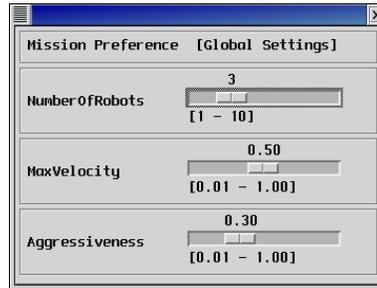


Figure 72: Mission Preference window to adjust the global setting.

- After the *mlab* session is closed, *CfgEdit* should bring up the window showing the summary of loadable mission plans based on the preferences and constraints you specified (Figure 73). You may browse them using and buttons. In this example, load the first mission plan being suggested.
- As you did in the previous demo, descend the level of the configuration until you see the FSA diagram, using the middle mouse button (Figure 74).
- Compile the mission by pressing the button in the left. Please note that if you happen to modify the suggested mission, Mission Expert may ask you to save the modified mission plan for the future use. In this case, toggle appropriate preferences and constraints buttons, and click on .



Figure 73: Summary of loadable mission plans with their suitability ratings.

7. Running the Generated Mission

- Click on the button in the left, and execute the mission.
- If your robots collect and dispose all the mines (orange objects) in the field (Figure 75), you have successfully created a mission using Mission Expert.
- After closing the *mlab* session, Mission Expert should ask you whether the mission was successful. This feedback will adjust the rating of the mission plan for the next time.

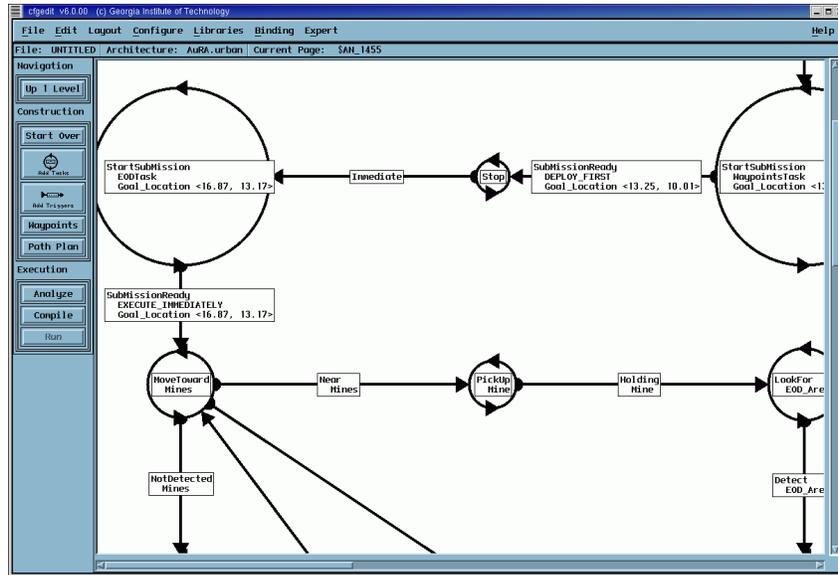


Figure 74: *cfgedit* showing the FSA of the generated mission.

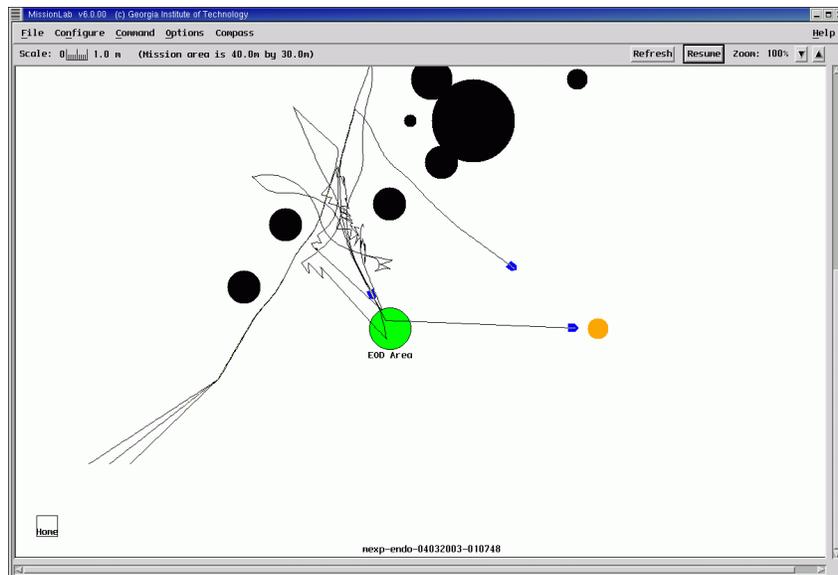


Figure 75: *mlab* showing the three robots executing the EOD task.

3.9.4 “Expert” Menu in *CfgEdit*

In *CfgEdit*, you may find the “Expert” menu (Figure 76), which can be used to invoke a few Mission Expert functionalities. If this menu is shaded out, check the *DisableMissionExpert* option in “.cfgeditrc” where the option should be set as “false”.

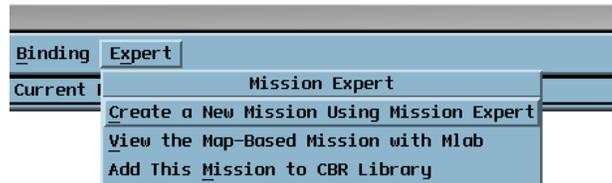


Figure 76: Mission Expert menu in *CfgEdit*

The first entry in this option is **Create a New Mission Using Mission Expert**. By selecting this option, you can build a new mission plan using Mission Expert as you did in Section 3.9.3. Make sure that *CBRServer* is started before selecting this option. The second entry **View the Map-Based Mission with Mlab** should be selected when you want to revisit the specs (preferences and constraints) you entered on the map using *mlab*. The third entry is **Add This Mission to CBR Library**. This option will send the current mission plan to *CBRServer*, so that it can be saved as a new case. It should be noted, however, that this option will not permanently store the mission plan in the CBR library file (“*.cbl”). In other words, if you terminate the *CBRServer* process, this mission plan will be erased. If you wish to save this mission plan in a CBR library file permanently, enter `s` key in the *CBRServer* window (See Section 3.9.2).

3.9.5 Repairing a Faulty Mission Using the Playback Function

One of the new capabilities added to the *CBR Wizard* since *MissionLab* 7.0 is the automated repair mechanism introduced in [15]. Here, the user can attempt to fix a mission that fails to achieved the user’s desired objective by retrieving a repair plan from the *CBRServer*. The example below shows how to use this capability.

1. Run the demo scrip

- Go to the demo directory.

```
% cd [your MissionLab home]/demos/usability_demos-2004
```

- Make sure there is an example CBR library file “usability_test-2004.cbl”.
- Run the script to start the *CBRServer* and *CfgEdit* processes.

```
% ./demo_cbr
```

2. Create a Biohazard Mission

- When *CfgEdit* asks, select “New Robot”, and “Yes” for *Mission Expert*.
- Select “factory.ovl” as the overlay.
- Set “MaxVelocity” to be 0.90.
- Click on the “Waypoints” icon in the toolbox.
- Place multiple WaypointTasks from *StartPlace* upto the *Factory Entrance*.

- Click on the “Biohazard” icon in the toolbox.
- Place BiohazardTask in the hallway near the entrance.
- Click on the “Finish” button in the toolbox.

3. Compile and run

- When the mission is executed, the robot will stop at the end of the hallway. Then, quit *mlab*.
- When asked, click “No” to a successful mission, and “Yes” to diagnosis.

4. Identify the faulty part of the mission

- *mlab* starts with the playback mode. Jog the robot until it reaches the end of the hallway (Figure 77).
- Click “Fix It!” in the toolbox.
- Click on “The robot did not perform a necessary action at this point”.
- Click on “The robot has to turn around at the end of the hallway”.
- Click “Yes” to apply the change.

5. Compile and run again

- In this time, the robot should turn around at the end of the hallway and successfully finds the biohazard at one of the rooms.

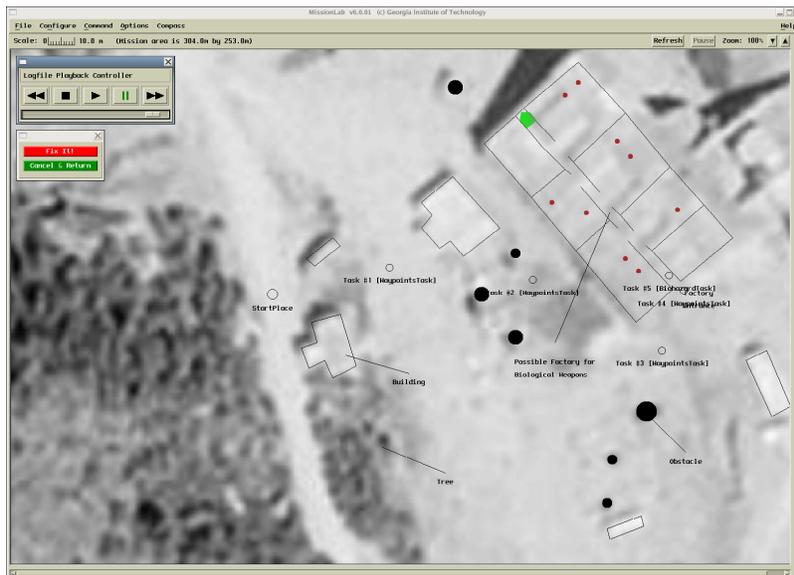


Figure 77: *mlab* - the playback mode

3.9.6 CBR-CNP Integration

Another new capability added to the *CBR Wizard* since *MissionLab 7.0* is the automated task-allocation functionality through Contract-Net Protocol (CNP). CNP is an auction-based negotiation algorithm that is used here to assign tasks to robots to achieve the overall mission. The task allocation may occur during the mission specification (premission-CNP) or during the execution (runtime-CNP). *MissionLab* supports both premission-CNP and runtime-CNP.

The following example shows how to specify a mission using the *CBR Wizard* while allocation of the tasks is done during the execution (runtime-CNP):

1. Run the demo scrip

- Go to the demo directory.

```
% cd [your MissionLab home]/demos/ao-fnc_demos
```

- Run the script to start the *CBRServer* and *CfgEdit* processes.

```
% ./demo_type-I-nmc
```

- Answer “New Robot”, “Yes” (to “Do you want to use Mission Expert?”), “Yes” (to “using CNP”), “No” (to “premission-CNP”), and “Yes” (to “Runtime-CNP”) when asked.

2. Create a Biohazard Mission

- Select “sea-mines.ovl” as the overlay.
- Set NMC (Naval Mine Countermeasure) as the overall mission, and hit “Apply”.
- Click anywhere on the overlay where you would like deploy the bidder robots.
- When the dialog pops up, click on the blue `Next >>` button until you see that `VEHICLE_TYPE` is USV (Unmanned Surface Vehicle).
- Hit the red “Select” button.
- Click again on the overlay and select another USV, so that there will be at least two bidder robots in the mission area.
- Click on the “Finish” button in the toolbox.

3. View, compile and run

- By descending down the configuration, take a look how the mission is composed.
- During the execution, the leader robot goes around to the spots where the mines are and starts an auction to allocate the mine-clearing job for the spot. The highest bidder among the worker robots will execute the job. In this case, bids are calculated based on the geographical location of the mine and robot.

3.9.7 Customizing Preference and Constraint Sets and Toolbox

Types of available preferences and constraints for Mission Expert are defined in `[your MissionLab home]/-src/cfgedit/.mission_expertrc` (where the symbolic link is also available in `[your MissionLab home]/bin`). An example of “.mission_expertrc” is shown at the end of this section. If you wish, you may customize this list.

To add a new feature (preference or constraint), its definition has to be enclosed within `<FEATURE BEGIN>` and `<FEATURE END>` clauses. `FEATURE-TYPE` option determines whether this preference or constraint should affect an entire mission plan (0) or just within a sub-mission (1). `FEATURE-NON-INDEX` specifies whether this feature should be treated as an index of the case for the case-based reasoning. If you would like it to be used as an index, set it to be 0. If it is used as the index for the case-based reasoning, `FEATURE-WEIGHT` specifies how much this feature should be weighted with respect to other features. `FEATURE-NAME` is the name of the feature. `FEATURE-OPTION-TYPE` specifies whether available values should be selected from discrete toggle buttons (0) or a continuous slider bar (1 for scaling from 0.10 to 1.00, 2 for scaling from 1 to 10). When the toggle button is used, each of the available values should be specified with `FEATURE-OPTION` (one value per `FEATURE-OPTION`). `FEATURE-SELECTED-OPTION` specifies the initially selected value (the first `FEATURE-OPTION` is 0, and the second one is 1, etc.). If the slider bar is used, the initially selected value should be specified with `FEATURE-SELECTED-OPTION`, and `FEATURE-OPTION` does not have to be specified.

The tasks in the toolbox (e.g., Biohazard, EOD, etc.) can be also customized in this file. The task definition has to be enclosed within <TASK BEGIN> and <TASK END> clauses. TASK-NAME sets the name of the task. TASK-BUTTON-NAME chooses which type of button to use. Currently, available button types are BiohazardBtn, MineBtn, OtherBtn. Additionally, the initially selected value specified with FEATURE-SELECTED-OPTION in the feature section above can be overridden with TASK-SPECIFIC-SELECTED-OPTION for this particular task. The name of the default task and its weight can be specified with DEFAULT-TASK and DEFAULT-TASK-WEIGHT, respectively.

```
#-----
# Example .mission_expert.rc
#-----
#
# FEATURE-TYPE (also used for CONSTRAINT-TYPE below)
#   CBR GLOBAL = 0
#   CBR LOCAL = 1
#   CBR LOCAL + CNP TASK-CONSTRAINT = 2
#   CNP TASK-CONSTRAINT = 3
#   CNP ROBOT-CONSTRAINT = 4
#   CNP RUNTIME-CNP (Mission Manager only) = 5
#   CNP RUNTIME-CNP (Bidder only) = 6
#   CNP RUNTIME-CNP (Mission Manager + Bidder) = 7
#   (Defined in mission_expert_types.h)
#
# FEATURE-NON-INDEX
#   FALSE = 0
#   TRUE = 1
#
# FEATURE-HIDE
#   FALSE = 0
#   TRUE = 1
#
# FEATURE-WEIGHT
#   Choose it from 0.00 to 1.00.
#
# FEATURE-OPTION-TYPE (defined in "mission_expert_types.h")
#   TOGGLE = 0
#   SLIDER1 = 1 ([0.10 - 1.00])
#   SLIDER10 = 2 ([1 - 10])
#   SLIDER100 = 3 ([1 - 100])
#
#-----

#-----
# Global Features
#-----

<FEATURE BEGIN>
FEATURE-TYPE 0
FEATURE-NON-INDEX 0
FEATURE-HIDE 0
FEATURE-WEIGHT 0.90
FEATURE-NAME NumberOfRobots
FEATURE-OPTION-TYPE 3
FEATURE-SELECTED-OPTION 1
FEATURE-OPTION N/A
<FEATURE END>
```

```
<FEATURE BEGIN>
FEATURE-TYPE 0
FEATURE-NON-INDEX 1
FEATURE-HIDE 1
FEATURE-WEIGHT 0.00
FEATURE-NAME MaxVelocity
FEATURE-OPTION-TYPE 2
FEATURE-SELECTED-OPTION 5.0
FEATURE-OPTION N/A
<FEATURE END>
```

```
<FEATURE BEGIN>
FEATURE-TYPE 0
FEATURE-NON-INDEX 1
FEATURE-HIDE 0
FEATURE-WEIGHT 0.00
FEATURE-NAME Aggressiveness
FEATURE-OPTION-TYPE 1
FEATURE-SELECTED-OPTION 0.25
FEATURE-OPTION N/A
<FEATURE END>
```

```
#-----
# Local Features
#-----
```

```
<FEATURE BEGIN>
FEATURE-TYPE 2
FEATURE-NON-INDEX 0
FEATURE-HIDE 0
FEATURE-WEIGHT 0.80
FEATURE-NAME ENVIRONMENT
FEATURE-OPTION-TYPE 0
FEATURE-SELECTED-OPTION 1
FEATURE-OPTION AIR
FEATURE-OPTION SURFACE
FEATURE-OPTION UNDERWATER
FEATURE-OPTION GROUND
<FEATURE END>
```

```
<FEATURE BEGIN>
FEATURE-TYPE 2
FEATURE-NON-INDEX 0
FEATURE-HIDE 0
FEATURE-WEIGHT 0.80
FEATURE-NAME MISSION_STEALTHINESS
FEATURE-OPTION-TYPE 0
FEATURE-SELECTED-OPTION 1
FEATURE-OPTION STEALTHY
FEATURE-OPTION NOT_STEALTHY
<FEATURE END>
```

```
#-----
# Runtime-CNP Features
#-----
```

```
<FEATURE BEGIN>
FEATURE-TYPE 7
```

```

FEATURE-NON-INDEX 1
FEATURE-HIDE 1
FEATURE-WEIGHT 0.80
FEATURE-NAME ROBOT_ID
FEATURE-OPTION-TYPE 3
FEATURE-SELECTED-OPTION 1
FEATURE-OPTION N/A
<FEATURE END>

<FEATURE BEGIN>
FEATURE-TYPE 5
FEATURE-NON-INDEX 0
FEATURE-HIDE 0
FEATURE-WEIGHT 0.80
FEATURE-NAME MISSION
FEATURE-OPTION-TYPE 0
FEATURE-SELECTED-OPTION 2
FEATURE-OPTION CARGO_INSPECTION
FEATURE-OPTION TARGET_TRACKING
FEATURE-OPTION NMC
FEATURE-OPTION VESSEL_INTERCEPTION
<FEATURE END>

#-----
# Robot Constraints
#
# ROBOT-OPTION-TYPE (Same as FEATURE-OPTION-TYPE.)
# TOGGLE = 0
# SLIDER1 = 1 ([0.10 - 1.00])
# SLIDER10 = 2 ([1 - 10])
#
#-----

<ROBOT-CONSTRAINTS BEGIN>
ROBOT-CONSTRAINTS-ROBOT-ID 1
<CONSTRAINT BEGIN>
CONSTRAINT-TYPE 4
CONSTRAINT-HIDE 0
CONSTRAINT-NAME VEHICLE_TYPES
CONSTRAINT-OPTION-TYPE 0
CONSTRAINT-SELECTED-OPTION 0
CONSTRAINT-OPTION UAV
CONSTRAINT-OPTION USV
CONSTRAINT-OPTION UUV
CONSTRAINT-OPTION UGV
<CONSTRAINT END>
<CONSTRAINT BEGIN>
CONSTRAINT-TYPE 4
CONSTRAINT-HIDE 0
CONSTRAINT-NAME STEALTHINESS
CONSTRAINT-OPTION-TYPE 0
CONSTRAINT-SELECTED-OPTION 1
CONSTRAINT-OPTION STEALTHY
CONSTRAINT-OPTION NOT_STEALTHY
<CONSTRAINT END>
<ROBOT-CONSTRAINTS END>

<ROBOT-CONSTRAINTS BEGIN>

```

```
ROBOT-CONSTRAINTS-ROBOT-ID 2
<CONSTRAINT BEGIN>
CONSTRAINT-TYPE 4
CONSTRAINT-HIDE 0
CONSTRAINT-NAME VEHICLE_TYPES
CONSTRAINT-OPTION-TYPE 0
CONSTRAINT-SELECTED-OPTION 2
CONSTRAINT-OPTION UAV
CONSTRAINT-OPTION USV
CONSTRAINT-OPTION UUV
CONSTRAINT-OPTION UGV
<CONSTRAINT END>
<CONSTRAINT BEGIN>
CONSTRAINT-TYPE 4
CONSTRAINT-HIDE 0
CONSTRAINT-NAME STEALTHINESS
CONSTRAINT-OPTION-TYPE 0
CONSTRAINT-SELECTED-OPTION 0
CONSTRAINT-OPTION STEALTHY
CONSTRAINT-OPTION NOT_STEALTHY
<CONSTRAINT END>
<ROBOT-CONSTRAINTS END>

<ROBOT-CONSTRAINTS BEGIN>
ROBOT-CONSTRAINTS-ROBOT-ID 3
<CONSTRAINT BEGIN>
CONSTRAINT-TYPE 4
CONSTRAINT-HIDE 0
CONSTRAINT-NAME VEHICLE_TYPES
CONSTRAINT-OPTION-TYPE 0
CONSTRAINT-SELECTED-OPTION 2
CONSTRAINT-OPTION UAV
CONSTRAINT-OPTION USV
CONSTRAINT-OPTION UUV
CONSTRAINT-OPTION UGV
<CONSTRAINT END>
<CONSTRAINT BEGIN>
CONSTRAINT-TYPE 4
CONSTRAINT-HIDE 0
CONSTRAINT-NAME STEALTHINESS
CONSTRAINT-OPTION-TYPE 0
CONSTRAINT-SELECTED-OPTION 0
CONSTRAINT-OPTION STEALTHY
CONSTRAINT-OPTION NOT_STEALTHY
<CONSTRAINT END>
<ROBOT-CONSTRAINTS END>

<ROBOT-CONSTRAINTS BEGIN>
ROBOT-CONSTRAINTS-ROBOT-ID 4
<CONSTRAINT BEGIN>
CONSTRAINT-TYPE 4
CONSTRAINT-HIDE 0
CONSTRAINT-NAME VEHICLE_TYPES
CONSTRAINT-OPTION-TYPE 0
CONSTRAINT-SELECTED-OPTION 3
CONSTRAINT-OPTION UAV
CONSTRAINT-OPTION USV
CONSTRAINT-OPTION UUV
```

```

CONSTRAINT-OPTION UGV
<CONSTRAINT END>
<CONSTRAINT BEGIN>
CONSTRAINT-TYPE 4
CONSTRAINT-HIDE 0
CONSTRAINT-NAME STEALTHINESS
CONSTRAINT-OPTION-TYPE 0
CONSTRAINT-SELECTED-OPTION 1
CONSTRAINT-OPTION STEALTHY
CONSTRAINT-OPTION NOT_STEALTHY
<CONSTRAINT END>
<ROBOT-CONSTRAINTS END>

<ROBOT-CONSTRAINTS BEGIN>
ROBOT-CONSTRAINTS-ROBOT-ID 5
<CONSTRAINT BEGIN>
CONSTRAINT-TYPE 4
CONSTRAINT-HIDE 0
CONSTRAINT-NAME VEHICLE_TYPES
CONSTRAINT-OPTION-TYPE 0
CONSTRAINT-SELECTED-OPTION 3
CONSTRAINT-OPTION UAV
CONSTRAINT-OPTION USV
CONSTRAINT-OPTION UUV
CONSTRAINT-OPTION UGV
<CONSTRAINT END>
<CONSTRAINT BEGIN>
CONSTRAINT-TYPE 4
CONSTRAINT-HIDE 0
CONSTRAINT-NAME STEALTHINESS
CONSTRAINT-OPTION-TYPE 0
CONSTRAINT-SELECTED-OPTION 1
CONSTRAINT-OPTION STEALTHY
CONSTRAINT-OPTION NOT_STEALTHY
<CONSTRAINT END>
<ROBOT-CONSTRAINTS END>

#-----
# Local Tasks
#
# TASK-BUTTON-NAME:
#   BiohazardBtn
#   MineBtn
#   WaypointsBtn
#   HostageMonitoringTask
#   SearchAndRescueTask
#   InspectBtn
#   InterceptBtn
#   MineOceanBtn
#   ObserveBtn
#   TrackBtn
#   OtherBtn
#
# TASK-MAX-USAGE
#   - The number of times this task can be placed.
#   - The value should be an integer or "inf" (for infinite number).
#
# TASK-RUNTIME-CNP

```

```
# - If the task is used for runtime-CNP
# - The value is 0 (false) or 1 (true). Default is 0.
#
#-----

<TASK BEGIN>
TASK-NAME ObserveTask
TASK-BUTTON-NAME ObserveBtn
TASK-SPECIFIC-SELECTED-OPTION ENVIRONMENT 0
TASK-MAX-USAGE inf
<TASK END>

<TASK BEGIN>
TASK-NAME EODTask
TASK-BUTTON-NAME MineOceanBtn
TASK-SPECIFIC-SELECTED-OPTION ENVIRONMENT 1
TASK-MAX-USAGE inf
<TASK END>

<TASK BEGIN>
TASK-NAME InterceptTask
TASK-BUTTON-NAME InterceptBtn
TASK-SPECIFIC-SELECTED-OPTION ENVIRONMENT 0
TASK-MAX-USAGE 1
<TASK END>

<TASK BEGIN>
TASK-NAME InspectTask
TASK-BUTTON-NAME InspectBtn
TASK-SPECIFIC-SELECTED-OPTION ENVIRONMENT 1
TASK-MAX-USAGE 1
<TASK END>

<TASK BEGIN>
TASK-NAME TrackTask
TASK-BUTTON-NAME TrackBtn
TASK-SPECIFIC-SELECTED-OPTION ENVIRONMENT 2
TASK-MAX-USAGE inf
<TASK END>

#-----
# Runtime-CNP Task
#-----

<TASK BEGIN>
TASK-NAME RuntimeCNPMissionManagerTask
TASK-BUTTON-NAME OtherBtn
TASK-MAX-USAGE inf
TASK-RUNTIME-CNP 1
<TASK END>

<TASK BEGIN>
TASK-NAME RuntimeCNPBidderTask
TASK-BUTTON-NAME OtherBtn
TASK-SPECIFIC-SELECTED-OPTION ROBOT_ID 2
TASK-MAX-USAGE inf
TASK-RUNTIME-CNP 1
<TASK END>
```

```
#-----  
# Other Miscellaneous Setup  
#-----  
  
DEFAULT-TASK ObserveTask  
DEFAULT-TASK-WEIGHT 1.00  
RUNTIME-CNP-MISSION-MANAGER-TASK RuntimeCNPMissionManagerTask  
RUNTIME-CNP-BIDDER-TASK RuntimeCNPBidderTask
```

4 Support Software

Several existing software packages were used in the development of *MissionLab*. These packages are described briefly here.

4.1 Communications Software IPT

Communication between the robot programs and the console program (*mlab*) is accomplished using the communication package IPT [7]. IPT is a transaction-oriented communication layer that sits above NSF sockets to simplify their use. As was described in Section 3.1, an IPT name server must be executing to enable the robots to find the console. *MissionLab* was developed with IPT version 8.4 but other versions of the IPT server will probably work.

4.2 Process Control Software C-Threads

The robot programs use the C-Threads process control software to control execution of the various sub-processes. C-Threads is a light-weight non-preemptive user level process scheduler package developed at Georgia Tech [17] to support multiprocessor computers. It allows development of multi-threaded programs which at run time can take maximum advantage of the available free processors. Support for inter-thread communication and synchronization mechanisms are also provided.

In the robot program, the Configuration Network Language compiler (the *cnl* compiler) instantiates each node (primitive behavior) as a separate thread of execution using the C-threads package. There exists the notion of a “cycle” where execution of each thread (node) occurs once and only once per cycle. This guarantees that the data flowing between nodes is synchronized. This means that within a single cycle, a node will not execute until all of the nodes providing it with input values have executed.

Currently, generated robot executables do not take advantage of multiple processors. That would be a simple extension from the compiler standpoint, but providing mechanisms for the designer to suggest groupings of nodes will require some thought. Good choices of groupings are necessary to reduce inter-nodal communication overhead.

5 Current Limitations of *MissionLab*

This is a preliminary version of the software and a number of features are not completely implemented. The following list describes some of the known limitations.

- When compiled on Fedora Core 5, *CfgEdit* Widgets may not be displayed properly if LessTif is used as the Motif package.
- If “Run” in the *CfgEdit* was selected, it runs with “mlab -R” option, whose pop-up window asks users to select an overlay to be used for the mission. The default overlay, which is automatically loaded before the pop-up window asks, is “Empty.ovl”, and this overlay has to be in the path “.cfgeditrc” specified. However, currently, users cannot choose multiple directories for the overlay location. In other words, there will be a problem if users want to select an overlay which is not in the same directory as “Empty.ovl”.
- When “mlab -R some_cmdl” was executed, *mlab* at first loads overlay which is in the cmdl file. (Let’s say, “default.ovl” was specified by cmdl file.) Now, because of this “-R” option, *mlab* will show the list of overlay files that are in the directory “.cfgeditrc” specified, so that users can chose the overlay which is suited for the mission. (Let’s say, the user wants “mission_A.ovl”.) The bug is that if the robot is supposed to start at StartPlace and there are different StartPlaces between “default.ovl” and “mission_A.ovl”, then *mlab* chooses the one in “default.ovl” instead of the one in “mission_A.ovl”.
- *mlab* cannot execute the next command in cmdl when it is running robots with the following situation: Suppose CarRobot, created by *CfgEdit*, moves back-and-forth on a road, and scout-robots trying to cross the road by going through multiple points. The first command is to start CarRobot and scout-robots, and the 2nd command is to move scout-robots to another point. The problem here is cmdl cannot get to the 2nd command because the CarRobot never stops. It needs some way to parallel process each “Command List”, so that the CarRobot can be executed in the background while scout-robots execute the commands in the other command list, sequentially.
- *mlab* has several serious memory bugs when it attempts to load command files, which may cause a crash. The bug erupts when trying to execute a “NEW-ROBOT” command without all the parameters listed. There is a workaround - specify the entire command line for the NEW-ROBOT command.
- The flexibility of the command description language is limited.
- The CMDL command FOLLOW is not implemented yet in the robot program.
- Bounding overwatch and traveling overwatch are not implemented.
- Only a limited set of overlay control measures are implemented.
- Control measures must be manually laid out in the overlay description file.
- Some polyline control measures (such as LD/LC) are currently drawn based only on the first and last points.
- Although phase lines can be given, they are ignored in the execution of mission plans.

6 FAQ / Trouble Shooting

Q. When I tried to compile MissionLab at first time, I got this error message. What was I doing wrong?

```
...building the IPT libraries...

cd ipt/src/communications/ipt; ./configure -unix LINUX -vx "" -install LINUX -main LINUX
mkdir: cannot create directory '/net/hr1/robot/mission/src/ipt/lib/LINUX': No such file or
directory
./configure: cd: /net/hr1/robot/mission/src/ipt/lib: No such file or directory
mkdir: cannot create directory '/net/hr1/robot/mission/src/ipt/include/ipt': No such file or
directory
./configure: cd: /net/hr1/robot/mission/src/ipt/include: No such file or directory
./configure: cd: /net/hr1/robot/mission/src/ipt/lib: No such file or directory
make install -C ipt/src/communications/ipt
make[1]: Entering directory '/net/hr1/robot/mission/src/ipt/src/communications/ipt'
g++ -g -DLINUX -I.. -c unixcommunicator.cc -o LINUX/unixcommunicator.o
unixcommunicator.cc:30: libc.h: No such file or directory
make[1]: *** [LINUX/unixcommunicator.o] Error 1
make[1]: Leaving directory '/net/hr1/robot/mission/src/ipt/src/communications/ipt'
make: *** [iptserver] Error 2
```

A. It usually happens when you did not specify a correct path in \$MLAB_HOME in [*your MissionLab home*]/src/make.include. Unfortunately, however, once you get this error, we found that the only way to recover from this is to start from the beginning, unpacking the *MissionLab* tar file. However, since *MissionLab* can now acquire the path for \$MLAB_HOME automatically when the configure script is executed during the installation, this problem should occur less frequently.

Q. I was able to compile a mission. However, when I try to run the mission, *mlab* pops up, but the robot never starts. The console leaves this error message. How can I fix this problem?

```

          IPT Client
          Version 8.3.0
Module Name: Console_15488
Server Host Name: localhost
Made connection IPT Server (localhost)
Attempting to execute defaultRobot1 -x 5.0 -y 5.0 -h 0.0 -i 1 -c Console_15488 -l columbus -t
localhost
Unable to exec the robot: No such file or directory
Unable to start robot 1 in send_robot_command
```

A. It usually happens when you did not set the execution path to “.” (e.g., .tcshrc). Read the section in the manual where it describes how to install *MissionLab*.

7 Feedback and Bug Reports

We welcome your feedback about this software. One of the purposes of releasing this software is to get some feedback from other researchers. We very much want to receive any suggestions, complaints, or questions about *MissionLab*. Theoretical and administrative questions should be addressed to:

Prof. Ronald Arkin
College of Computing
Georgia Institute of Technology
801 Atlantic Dr.
Atlanta, GA 30332-0280
arkin@cc.gatech.edu

If you encounter a bug, please document it as completely as possible and send that information to us via email to m1ab@cc.gatech.edu.

References

- [1] Arkin, R.C. *Behavior-Based Robotics*. Cambridge, the MIT Press, 1998.
- [2] Arkin, R.C. "Motor Schema-Based Mobile Robot Navigation", *International Journal of Robotics Research*, Vol. 8, No. 4, August 1989, pp. 92-112.
- [3] Arkin, R.C., Clark, R.J., and Ram, A., "Learning Momentum: On-line Performance Enhancement for Reactive Systems," Proceedings of the 1992 IEEE International Conference on Robotics and Automation, May 1992, pp. 111-116.
- [4] Arkin, R.C., Collins, T.R., and Endo, Y. "Tactical Mobile Robot Mission Specification and Execution." *Proceedings of SPIE*. Vol. 3838 (Mobile Robots XIV). 1999
- [5] Balch, T. and Arkin, R.C., "Avoiding the Past: A Simple but Effective Strategy for Reactive Navigation", Proc. 1993 IEEE International Conference on Robotics and Automation, Atlanta, GA, May 1993, Vol. 1, pp. 678-685.
- [6] Balch, T. and Arkin, R.C. "Communication in Reactive Multiagent Robotic Systems," *Autonomous Robots*, Vol. 1, 1994, pp. 1-25.
- [7] Fedor, C. "TCX: Task Communication," (User manual for TCX, available through the Robotics Institute), Carnegie Mellon University, Feb. 15, 1993.
- [8] Koenig, S. and Likhachev, M. "Improved Fast Replanning for Robot Navigation in Unknown Terrain." In Proceedings of the International Conference on Robotics and Automation, 2002.
- [9] Lee, B., and Hurson, A.R. "Dataflow Architectures and Multithreading," *IEEE Computer*, August 1994, pp. 27-39.
- [10] Lee, J. B., Arkin, R. C., "Learning Momentum: Integration and Experimentation," Proceedings of the 2001 IEEE International Conference on Robotics and Automation, May 2001, pp. 1975-1980.
- [11] Likhachev, M., and Arkin, R.C., "Spatio-Temporal Case Based Reasoning for Behavioral Selection," Proceedings of the 2001 IEEE International Conference on Robotics and Automation, May 2001, pp. 1627-1634.
- [12] Likhachev, M., Kaess, M., and Arkin, R.C., "Learning Behavioral Parameterization Using Spatio-Temporal Case-Based Reasoning," Proceedings of the 2002 IEEE International Conference on Robotics and Automation, May 2002, pp. 1282-1289.

-
- [13] 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.
 - [14] MacKenzie, D.C. and Arkin, R.C. "Formal Specification for Behavior-Based Mobile Robots," SPIE Mobile Robots VIII, Boston, MA, November 1993.
 - [15] Moshkina, L., Endo, Y., and Arkin, R.C. "Usability Evaluation of an Automated Mission Specification," Proc. First Human-Robot Interaction Conference, Salt Lake City, 2006.
 - [16] Piaggio, M., Sgorbissa, A., and Zaccaria, R., "Micronavigation", Proc. Sixth Int. Conf. on Simulation of Adaptive Behavior, SAB2000, Paris, 2000
 - [17] Schwan, K., et. al. *A C Thread Library for Multiprocessors*, Georgia Tech Technical report #GIT-ICS-91/02, 1991.
 - [18] Sgorbissa, A. and Arkin, R.C. "Local Navigation Strategies for a Team of Robots", Georgia Tech Technical report #GIT-ICS-00/05, 2000.