

Manual for ADE
For CSE498F: Behavior Based Robotics
Spring 2004

Virgil Andronache
Department of Computer Science
University of Notre Dame
USA
vandrona@cse.nd.edu

Trevor Cickovski
Department of Computer Science
University of Notre Dame
USA
tcickovs@nd.edu

February 12, 2004

Chapter 1

Setting Up ADE

1.1 Using the correct java

A java executable that includes all features of the latest developer pack, specifically the necessary libraries for XML parsing has been placed in `/usr/local/src/cse/java/j2sdk1.4.2_03/`. This java is the one that should be used for running all Java classes in the scope of this course. It thus may be helpful to add the following line to your `.cshrc` file:

```
alias java '/usr/local/src/cse/java/j2sdk1.4.2_03/bin/java'
```

This way the correct java will always be used upon logging in to a system by simply typing `'java'`.

Also, specifically those students who took Intro To E-Technologies in the fall or have in the past may have set macros `JAVA_HOME` and `CATALINA_HOME` in their `.cshrc` files. If these macros are set to anything in your `.cshrc` they must be commented out or deleted. Otherwise these macros and the current java being used will likely conflict and result in strange errors.

1.2 Setting up a .apoc/apoc.ini file

The next step is to create a `.apoc/` directory in the root directory of your personal AFS space. Within that directory you will need to place one file called `apoc.ini` which will automatically be sourced upon your running of the ADE server. A registry must be started on a specific machine in order to start the ADE server, though the simulator itself can be run on several different machines. The role of the registry is basically a system monitor: It keeps track of what servers are running in the system and the services they provide. Thus it also acts as a broker (e.g., if there is a vision server and someone wants to use it, they request a connection to the server through the registry). You may choose any machine to start the registry on, but preferably one where there is not a current heavy load. We should have a registry running continuously on `airolab.cse.nd.edu`, but for now please start one every time you run programs, by running:

```
java -classpath /afs/nd.edu/courses/cse/cse498f.01/cse498f.jar com/AgeSRegistryImpl
```

Here is an example `apoc.ini` file with a registry running on the machine `larson.helios.nd.edu` on the first floor Fitzpatrick lab, and simulators running on `larson` and `bronte` (which is close by):

```
REGISTRY = larson.helios.nd.edu AgeSRegistry$AgeSRegistry
```

```
SSH = /usr/local/bin/ssh
SSH_COMMAND = /afs/nd.edu/user32/vandrona/testfile
HOSTS = laron.helios.nd.edu bronte.helios.nd.edu
UNITDIRS = apoc/units
OPERATOR_DIRS = apoc/operators
INPUTFILE = /afs/nd.edu/user32/vandrona/nets/nnAExample.net
```

with the `REGISTRY` macro defining the one machine to start the registry on, and the `HOSTS` macro specifying which machines to run the simulators. Since a necessary step in the running of servers is to connect to each of these `HOSTS` machines via secure shell, it is necessary to specify the command for `ssh` using the `SSH` macro. The `SSH_COMMAND` macro supplies a filename in which the appropriate commands are placed to start the server on the various `HOSTS`. The `testfile` used for the above example of `larson` and `bronte` contains the following:

```
java -classpath /afs/nd.edu/courses/cse/cse498f.01/cse498f.jar com/apoc/APOCServerImpl
larson.helios.nd.edu &
```

A line of the form

`java com/apoc/APOCServerImpl <registry_hostname>` is all that is needed. `UNITDIRS` contains the set of directories where the program will look for units (modules) to build an architecture from, and `OPERATOR_DIRS` is for links in the architecture, which will modify data going through it allowing the user to transform the data passing through a link in the architecture in some specific way conducive to the design. If all of this does not make complete sense now, it will be more clear in the section on *Running ADE*. The final macro, `INPUTFILE`, is currently unused but may be in the future so it is there but for now you may disregard it.

The one final step is to ensure that you have an `ssh` agent running in the terminal where you plan to start your simulation. This done with a sequence of two commands:

```
eval `ssh-agent`
ssh-add
```

On the second you will get asked for a passphrase, just hit enter and you should be fine.

For more information in `ssh` and `rsa` keys, check out: <http://acd.ucar.edu/fredrick/mpark/ssh/rsa-unix.html>

Chapter 2

Running ADE

The registry is started with `java com/AgeSRegistryImpl`. Then while the registry is running, on each machine where you want to run the simulator type `java APOCstart`. You will need to have an ssh agent running in each of these terminals as described in the first section. This will bring up a GUI, and you are then ready to load your classes.

The following sections outline the menus and buttons that you will need for the course, and their functionality.

2.1 Menus

2.1.1 File Menu

- **New:** Clear the current framework.
- **Open:** Insert an XML architecture description into the framework.
- **Save/Save As:** Save the current framework as an XML file.
- **Shutdown:** Stop the current simulator (but no others that may be networked)
- **Quit:** Quit the entire simulation. All networked simulators close.

2.1.2 Edit Menu

- **Random Seed:** The seed for random values, should not need to be changed.
- **Preferences:** Allows you to set the search path for component units to be added. You may add or remove directories from the search path, as well as display the current path as it stands.
- **Permissions:** Allows control over what can be done with the system - should not need modification.

2.1.3 View Menu

This will hide or view various parts of the architecture. For now, do not worry about this, just keep everything visible (the default).

2.1.4 Mode Menu

Switch between simulation edit and run mode. Each of these are described in more detail later.

2.1.5 Network Set-up Menu

This only applies to a simulation run over a network. Control is given here over which hosts are to be used for the simulation. You may add, remove, show or reset your available hosts for the simulation, or set the host for the registry itself. One registry on one machine is associated with each simulation.

2.1.6 Help

This menu is self-explanatory.

2.2 Edit Mode

2.2.1 Description

An architecture framework is set up in edit mode. This is done once units have been coded in Java, compiled and placed somewhere in the units path (mentioned earlier in the “Menus” section). You will have the responsibility of defining your own units and their capabilities throughout this course. Units can be added to the framework as well as links between them. This is all accomplished through the “Virtual Machine View” in edit mode, and the following buttons:

2.2.2 Buttons

- **Add Unit:** Search the units path and display all possible units, add one to the framework upon selection.
- **Add Link:** Add a communication link between two units.
- **Add Label:** Place a specific label on a unit in the Architecture View.
- **Zoom In/Out, Pan:** Controls the field of vision in the Architecture View.
- **Arrow:** Get back to a pointer.
- **Group:** Highlight a group of components.
- **Toggle:** For specific values, toggle between + and -.
- **Delete:** Delete the selected component/group of components.
- **Ungroup:** Un-highlight a group of components.
- **Copy:** Copy the selected component/group of components to the clipboard.
- **Paste:** Paste component(s) on the clipboard at a desired location.
- **Undo:** Undo your most recent action.

2.3 Run Mode

2.3.1 Description

Run mode is used once an entire framework has been loaded using edit mode and is ready to run. A simulation using this software is composed of a certain number of cycles - that is, one cycle amounts to a certain timeslice where the robot performs certain actions “updating itself”, then goes on to the next cycle. There is a box available in run mode which accepts an integer for the number of cycles to run. The buttons control the rest.

2.3.2 Buttons

- **Update:** Run the simulation for the specified number of cycles.
- **Cycle:** Run the simulation for one cycle.
- **Reset:** Reset the simulation to cycle 0.
- **Abstraction:** Don't worry about this for now.
- **Instantiate:** This initializes the simulation appropriately. It must be run in the beginning before updating or cycling.
- **Graph:** Allows you to click on a node in the simulation and show how some variables have changed over time.

Chapter 3

Working with the robots

Two steps need to be taken in order to work with the robots. First, log on using your afsid and the last four digits of your SSN as a password (change the password after you log in). Then cd to /usr/local/cse498f and run the following commands:

1. `java com/pioneer/PioneerServerImpl ;registry_host<. This starts up the robot (you should hear the sonars clicking) and connects it to the registry. You can now connect to the robot through the registry by requesting a "Pioneer". This is taken care of in the code provided in the file $$$java. The pioneer is then accessed through the robot variable.`

This gives you access to the following useful functions, among others:

```
float robot.myBatterySensor.getVoltage();
```

This tells you the voltage of the battery

```
boolean robot.myBumperSensor.getBumper(int w);
boolean[] robot.myBumperSensor.getBumpers();
```

The robot has 10 bumpers, which you can read individually, or in an array.

```
int[] robot.myMotorSensor.getEncoders();
```

This tells you how far the robot has moved, in encoder counts for each wheel

```
float[] robot.mySonarSensor.getSonars();
float robot.mySonarSensor.getSonar(int w);
```

The robot has 16 sonars, which you can read individually, or in an array. Readings are in millimeters.

```
void robot.myMotorEffector.turn(int magnitude);
void robot.myMotorEffector.turnTo(int position);
void robot.myMotorEffector.setVelocity(int vel);
void robot.myMotorEffector.setWheelVelocity(int[] vel);
void robot.myMotorEffector.go(int distance);
void robot.myMotorEffector.stop();
void robot.myMotorEffector.emergencyStop();
```

2. `java -Djava.library.path=/usr/local/v4l com/framegrabber/FramegrabberServerImpl <registry_host>.` This starts the framegrabber, which allows you to get images from the camera. To get an image, call

```
<variable\_name>.getCameraPicture(int quality);
```

This returns a jpeg image, with the quality specified. Conversion from jpeg to the java `BufferedImage` format and viceversa is done as below:

```
public static BufferedImage jpegToBufferedImage(byte[] pic) {
    if (pic == null)
        return null;
    ByteArrayInputStream JPEGstream = new ByteArrayInputStream(pic);
    JPEGImageDecoder JPEGDecoder = JPEGCodec.createJPEGDecoder(JPEGstream);
    try {
        return JPEGDecoder.decodeAsBufferedImage();
    } catch (IOException e) {return null;}
}

public static byte[] bufferedImageToJPEG(BufferedImage bi) {
    byte[] returnImage;
    ByteArrayOutputStream JPEGGout = new ByteArrayOutputStream();
    JPEGImageEncoder JPEGEncoder = JPEGCodec.createJPEGEncoder(JPEGGout);
    try {
        JPEGEncoder.encode(bi);
    } catch (IOException ioe) {
    }
    returnImage = JPEGGout.toByteArray();
    return returnImage;
}
```

Note that these require you to put *import com.sun.image.codec.jpeg.*;* at the top of your file.

3.1 Other facilities

Blob and motion detection functions are provided with the system in files `agent.pioneer.brains.brain1.BlobDetectionNR` and `agent.pioneer.brains.brain1.MotionDetectionNR`. Instances of both classes are simply initialized as

```
BlobDetectionNR bd = new BlobDetectionNR();
MotionDetectionNR md = new MotionDetectionNR();
```

To set the colors which are to be identified as parts of blobs, by the `BlobDetection` class, use

```
bd.storeColors(blobRanges);
```

`blobRanges` in this case is a double array of integers, representing sets of RGB values, in that order

```
int[][] blobRanges = new int[][]{{85, 235, 150, 230, 210, 255}, {45, 100, 95, 145, 25, 245}};
```

Blob detection is then done simply as:

```
blobData = bd.getBlobs(bi);
```

where `bi` is a `BufferedImage`. The return value is a `Vector` of `Vectors`. Each of the individual vectors holds a set of blobs for the respective RGB range (in the above example, `blobData` would be a `Vector` of 2

Vectors). Each Blob data structure has `xcg` and `ycg` coordinated, which give you the centroid of the blob, as well as an `area`, which gives you the total number of pixels in the blob.

For calibration purposes (to find out what ranges work best), you can run

```
java -classpath cse498f.jar com/camview/CamviewServerImpl <registry_name>
```

This will bring up a GUI on which you can use sliders to set the ranges for blob detection.

Motion detection works similarly:

```
motionBlobs = md.getBlobs(bi,oldBI,false);
```

where the first two parameters are the current and last pictures received from the camera. The return is a Vector of Blobs.

To view how this works, you can use the same GUI as for blob calibration.

3.2 Closing Remarks

We both invite anyone with further questions about the software or things covered in the manual to come in contact with us.