Reinventing the Reinvented Shakey in SNePS

SNeRG Technical Note 36

Trupti Devdas Nayak
Michael Kandefer
Lunarso Sutanto
Department of Computer Science and Engineering
University at Buffalo, The State University of New York
Buffalo, N.Y. 14260-2000

{td23 | mwk3 | lsutanto}@cse.buffalo.edu

April 6, 2004

Abstract

This paper describes the implementation of the Shakey robot on a Magellan Pro hardware robot. The paper gives an account of the SNePS/GLAIR architecture which has been implemented for the Magellan Pro. It details a challenging task of reinventing Shakey in order to expand the horizons he has inhabited so far. The aim is to utilize prior knowledge about SNePS, build a GLAIR architecture and integrate this with the capabilities of the Magellan Pro so that we have a robot capable of sensing, planning and acting intelligently. The paper also describes in detail the Knowledge Level of the GLAIR architecture which is implemented in SNePS for the Magellan Pro hardware robot.

Contents

1. Introduction

Shanahan [5] describes in his “Reinventing Shakey” paper, an implementation of the Shakey robot using event calculus [7] and a Khepera robot. The Shakey problem is described as a robot placed in a closed environment containing rectangular rooms and doors which open into adjacent rooms. In this environment there is also a package and the robot’s goal is to calculate a path to this package from the robot’s starting location. It then makes its way to the package using wall-following and turn-at-corner functions, and determines its state in the world using sensors like sonar sensors. In order to do this, the robot must have some internal representation of the world, or access to one. One problem which exists is that a door can be shut or opened at any time, and if the robot’s original path doesn’t work, it is up to the robot to decide which door was closed, and to calculate a new path from its present location.
The goal of our project is to implement a Shakey robot using SNePS and iRobot’s [8] Magellan Pro hardware robot. We are using a substantially smaller environment, two rooms and one door connecting them, due to the size of our robot and our limited movement space. A subsequent goal of the project is to develop this paper, detailing the steps we took to accomplish this goal, and explaining our experiences with the systems used, which are new to us.

2. SNePS and the GLAIR architecture

“SNePS (the Semantic Network Processing System) is a system for building, using and retrieving information from propositional semantic networks.” [1]. SNePS is basically used for knowledge representation at various levels. It is called ‘propositional’ because every proposition is represented by a node in the semantic network. The arcs in the network represent relations between the nodes they connect. We also need to construct the **GLAIR** (*Grounded Layered Architecture with Integrated Reasoning*) for the Magellan Pro robot. The GLAIR architecture consists of three layers [3] which is illustrated in Figure 1.

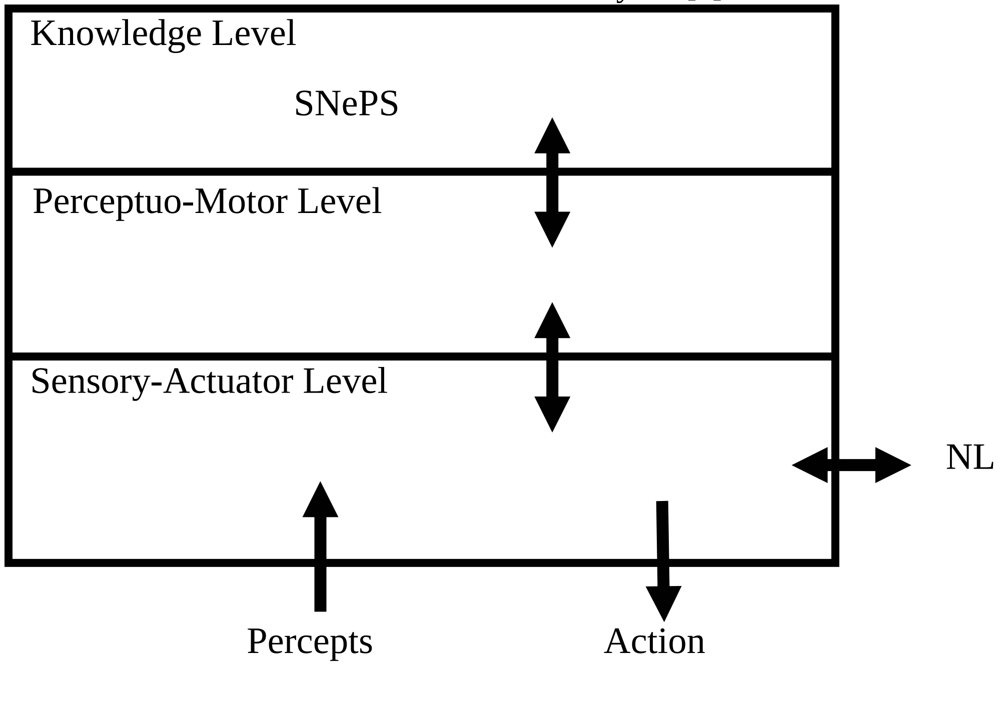

Figure 1: GLAIR Architecture[4]

2.1 Knowledge Level

The KL has the knowledge base of the robot at the abstract level. It consists of what the robot is consciously aware of. At this layer, the knowledge is represented using SNePS and its sub-system SNeRE so that actions can be executed by the robot based on what it believes and perceives. SNePS is implemented in Common Lisp.

2.2 Perceptuo-Motor Level

The PML has the physical level representations of the objects. It has the most primitive actions. The PML has 3 sub-levels [3]:

2.2a *PMLa*

The PMLa is the highest level where the ideas that KL represents are translated into functions. This level has the functions which will be used for implementing the KL level. There are a set of primitive actions which determine the basic moves the robot is able to accomplish. These functions are implemented in SNePS. These include the follow-wall and go-straight actions performed by the robot.

2.2b *PMLb*

The PMLb plays a role in interfacing the PMLa and the PMLc. Its role is fuzzy in this particular project since it is combined with PMLa and with PMLc. A mention of it is made since the PML of the GLAIR architecture consists of 3 sub-levels.

2.2c *PMLc*

The PMLc level has been implemented in C++. It consists of the server program which will execute on the robot. The C++ program has a set of functions which take input from the program running on a client and depending on the task, the robot behaves accordingly. The sensor functions access the hardware and receive the sonar values. Based on predetermined threshold cut-off values, the variables front-high, front-low, left-high and left-low are set. These variables are accessed by the higher PML layers and functions like move-front, move-behind, turn-left and move-forward.

2.3 Sensory-Actuator Level

The SAL consists of the actual sonar sensors which determine what the robot perceives. This input is given to the higher layers, and the plan made accordingly. The SAL can consist of any other hardware/software devices which enable perception in the intelligent entity with respect to the environment.

3. Introduction to Our Shakey

The Shakey project started in the late sixties [11], using a logic based approach. The idea for constructing Shakey was driven by the desire to provide a high-level cognitive implementation of a robot which is capable of mental skills attributed solely to human beings, such as planning, reasoning about oneself, reasoning about other agents, and communicating with other agents [5]. The Shakey project was revived and implemented for a Khepera [5] which is a real robot, and inhabits an environment depicted in Figure 2 which is a prototype for the environment that Magellan will inhabit which is depicted in Figure 3.

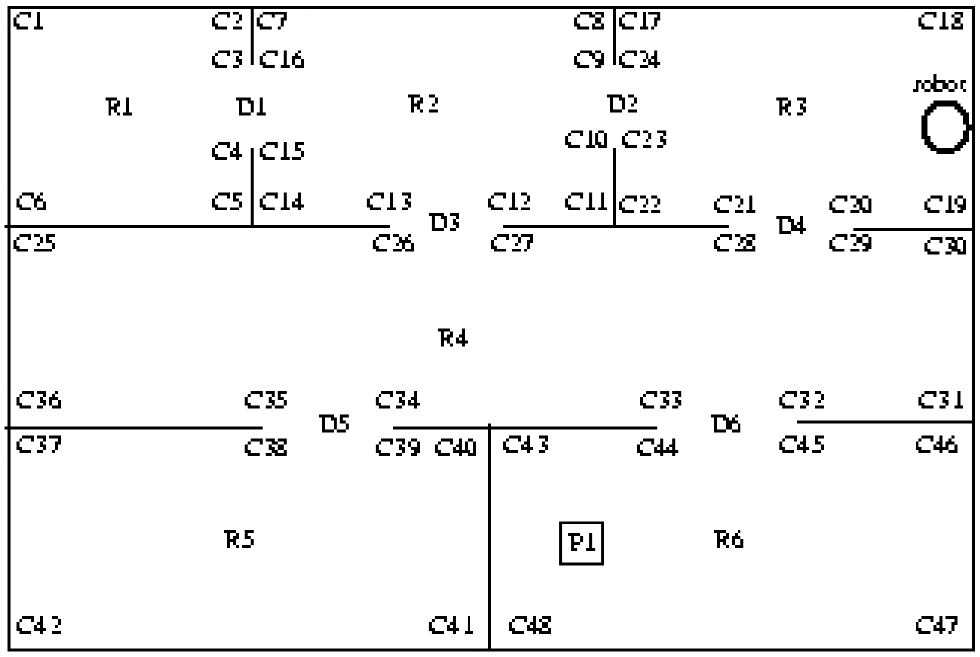

Figure 2: Shakey's original environment [5]

The environment we will be adapting is a simpler version of the one shown in Figure 2. It is deliberately kept simple, because this will enable us to focus on implementing it for a much larger robot than the Khepera, and once this has been accomplished, the complexity can be incrementally increased.

3.1 Magellan/Shakey's environment

As of December 2003, Shakey operated in a basic environment consisting of 2 rooms shown in Figure 3. Though the room was not large enough to have as many divisions as the original Shakey environment, it sufficed for our needs.

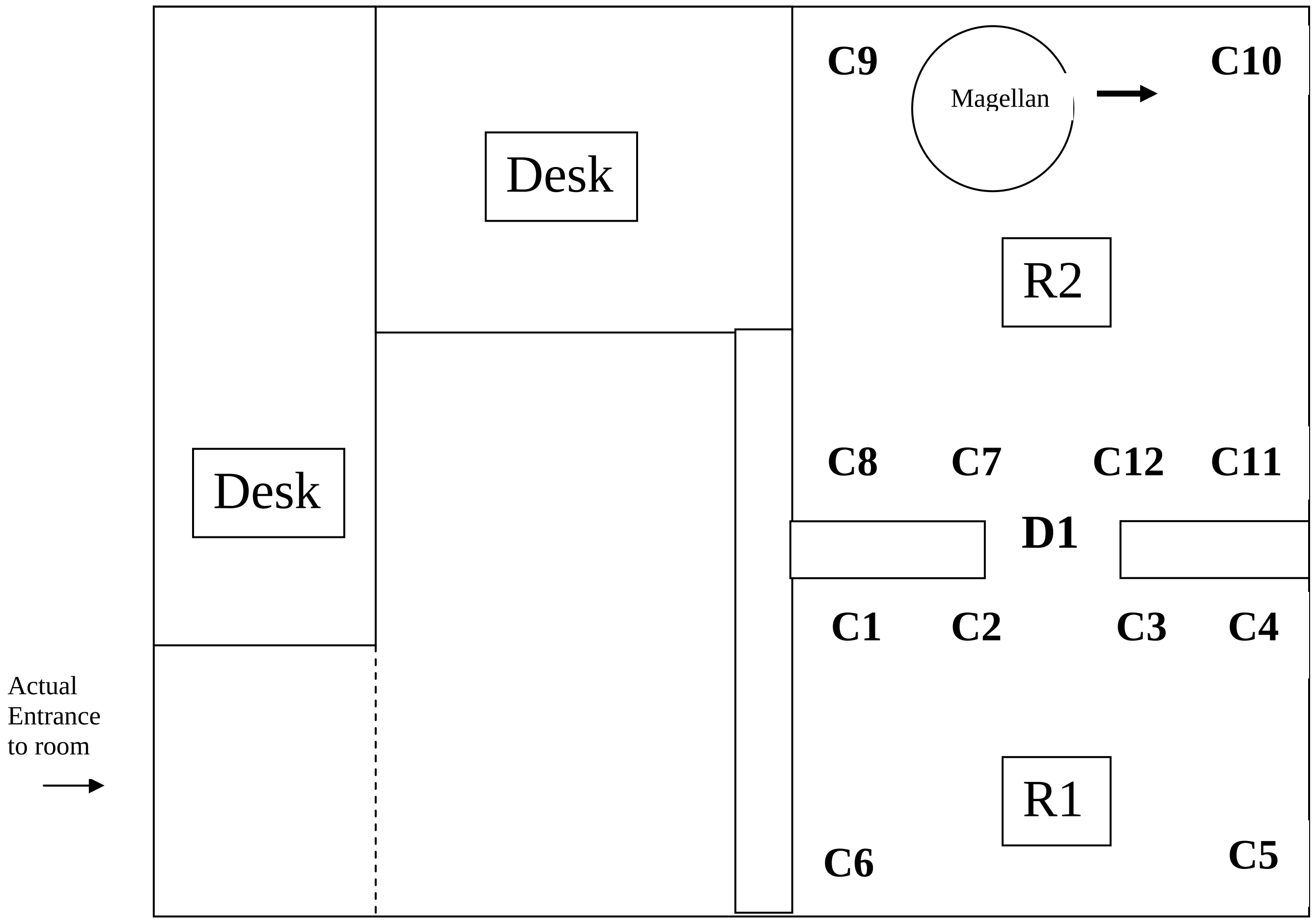

Figure 3: *Map of Room 329, Bell Hall*

Legend:
(1) R1, R2 ----- Rooms
(2) C1, C2, C3, C4, C5, C6 ----- Corners of R1 (where C2, C3 are outer corners)
(3) C7, C8, C9, C10, C11, C12 ----- Corners of R2 (where C7, C12 are outer corners)
(4) D1 --- Door connecting R1 and R2
(5) Magellan ---- alias Shakey

The Shakey robot is assumed to move beside the left wall (follows the left wall). The arrow in front of the robot in Figure 3 depicts the direction of movement. Our aim is to use Shakey's sensing capabilities and reasoning power to make it get out of R2 (room 2) into R1 (room 1). In order to do this, Shakey has to travel from corner to corner of the room, detecting these corners, detect a door if one is encountered, and go through the door to get into the second room.

The following should be noted:

1. The room is small. Hence, the speed of the robot has to be maintained at a manageable level. A threshold has to be carefully chosen so that the robot can take immediate action the moment it detects it is now approaching a corner of the room.

2. The robot is self-aware. At any given point of time, Shakey should be aware of its position in the room. If it is 'lost', it will remain lost because it will not 'know' where the door is or which corner it is currently at.

3. As an addition to its repertoire of functions, one function proposed to be implemented for Shakey will determine if it is moving away from the wall it is following. Shakey only follows the left wall, so if for some reason (like, being a medium sized hardware robot, if the wheels are not aligned well) it starts moving away from the wall, the sonar reading will be incorrect and the robot has no idea of where it is.

4. Aims for Magellan

The main current aim for Magellan is to move along the left wall, and turning right when a corner is encountered, and getting to room R1 through the connecting door D1.

The sub tasks Magellan should be able to achieve for the above aim are

1. Move forward, backward as the user instructs
2. Turn right when he detects a corner.
3. When a door is detected through which he has to go, as part of the plan, he turns left at the outer corner and enters the other room.
4. He stops when the door is closed and does not keep circling the room.(this is the sign of an intelligent robot. The robot can determine this because it has a map for reference. When it expects a door to be present, but instead of passing a door, it reaches a corner, it can safely assume that it passed the door, and the reason why it did not observe the door was because it was closed. The robot's sensors are weak, and when the door is closed, it cannot differentiate between the wall and the door.

5. Implementing Shakey through Magellan

This gives an overview of the actual implementation process, involving design, coding and testing.

5.1 Introduction to Magellan Pro and the MOM Interface

5.1.1 Magellan Pro

The Magellan Pro was developed by iRobot and is a recent addition (Fall 2003) to the University at Buffalo's Computer Science and Engineering Department. Our Magellan Pro is equipped with several sensors: sixteen tactile sensors, used to determine if the robot has collided with an object, sixteen sonar sensors, used to determine distances from objects, and a camera for vision processing that can pan or tilt. As for actuators the robot is slightly limited in that it only comes with the ability to move forward, backward, and rotate right or left with its two wheels underneath the base. However, since the robot has an internal PC running Linux one is capable of adding their own hardware to the unit, as long as the robot can support the new hardware's weight. Controlling both the sensors and actuators can be done in two ways: either through user control or program control. Programming in the Magellan robot is done using the mobility package, which provides various data structures and functions for accessing the three services our robot provides; base actuators, base sensors, and the camera. The mobility package is implemented using CORBA and C++, though a great deal of CORBA knowledge isn't needed to program the robot.

5.1.2 MOM Interface

The Magellan package also includes a software application entitled the MOM (Mobility Object Manager) Interface. The MOM Interface allows users of Magellan to acquire data about the various robot sensors and control the motors of the robot. For the Magellan robot this means one can view the tactile, sonar, and camera sensors dynamically. This is both effective for debugging, and acquiring threshold values. One can also use the MOM information to control the Magellan's movements and the pan/tilt functionality of the camera. This allows one to navigate an environment getting an idea of what the robot will need to do before jumping straight into programming. Once the MOM interface is running, accessing these features is easy. By clicking a few drop-down menus and right clicking on the sensor or actuator one can read or manipulate these features. Since the Mobility manual [9] covers this nicely we'll leave it to the reader to research how. However, there are a few steps that must be taken first before the MOM interface can be started.

5.1.3 Getting started

There is a Unix machine which has SNePS (Lisp) installed on it. The operating system on the robot has the software required to run it. The user logs into the robot server using the unix machine. The robot has to be powered on in order to be able to log in. After logging in, the server and the client programs can be compiled and loaded.

Log into the system from which the robot can be accessed. The following example (Comments are in parenthesis) is with Michael Kandefer's login name and directory path using csh.

```
(Remote login to the robot itself using ssh.)
[mwk3@oldred ~]> ssh irobot.cse.buffalo.edu

(Source the mobility package.)
[mwk3@irobot ~]$ source ~mobility/mobility.csh

(Start the name service.)
[mwk3@irobot ~]$ name -i

(Start the sensor/actuator servers.)
[mwk3@irobot ~]$ startup

(Start MOM interface.)
[mwk3@irobot ~]$ mom
```

5.1.3.1 Server on Magellan

The program running on Magellan is the server program which waits for input through sockets. The sockets connect it to the client program running on the Unix machine. The server on Magellan is written in C++ with CORBA objects which implement the robot motion commands. The task of the server is to parse the strings sent down the socket by the

Lisp/SNePS program. After parsing, the C++ program gets the primitive action which has to be executed.

5.1.3.2 Client on Unix machine

The client is on the Unix machine because it is requesting the server for an instruction to be executed. This is keeping in view of the fact that in the future, there will be many clients on other machines, sending the server on the robot requests for achieving an aim. The client program was initially written as a Lisp program. A corresponding SNePS program uses the functions defined in the Lisp program.

5.1.3.3 Socket Programming

Sockets are the only means of communicating strings between the different processes. Multiprocessing, Semaphores and Networking with ACL [6] is an extremely useful and informative paper which provided information on creating a socket, sending strings across and closing the socket. It also has information on running threads in the background which might be needed if more than one program will be sending strings down the sockets to the server running on the robot.

6.2 Programming Shakey in the lower levels

6.2.1 Accessing sensors and actuators with Mobility (SAL/PMLc)

One of our primary tasks for this project is to program the low level systems. To program Shakey in the lower levels one has to know how to access the various hardware components (PMLc/SAL) and C++/CORBA. What follows are the steps necessary and the program used (cserver.c++) to access the various hardware components. It is important to note that the interface must be activated before a program will execute correctly; this is to say the name service and sensor/actuator servers. In addition to this the appropriate libraries must be included in the program that will manipulate the robot. This is done as follows:

```
// These includes pull in interface definitions and utilities.
// This goes at the top of the PMLc C++ file

#include "mobilitycomponents_i.h"
#include "mobilitydata_i.h"
#include "mobilitygeometry_i.h"
#include "mobilityactuator_i.h"
#include "mobilityutil.h"
```

Once the appropriate libraries are included, all that remains before the hardware components can be accessed is to initialize the CORBA C++ initialization and acquire access to the robot server(s) we are interested in accessing. Once setup, we can create the data structures for accessing the robot’s sensors and actuators. Since our robot only uses sonar sensors and the wheel actuators we will only discuss these. This is done as follows (some code segments below come from the Mobility manual [9]):

```
// The following code comes from the PMLc C++ file on the robot.
// This framework class simplifies setup and initialization for
// client-only programs like this one.
mbyClientHelper *pHelper;

// This is a generic pointer that can point to any CORBA object
// within Mobility.
CORBA::Object_ptr ptempObj;

// Used for accessing the robot’s actuators (wheels)
MobilityActuator::ActuatorState_var pDriveCommand;
MobilityActuator::ActuatorData OurCommand;

// Used to access the sonar sensor data
MobilityGeometry::SegmentState_var pSonarSeg;
MobilityGeometry::SegmentData_var pSegData;

// All Mobility servers and clients use CORBA and this initialization
// is required for the C++ language mapping of CORBA.
pHelper = new mbyClientHelper(argc,argv);

// Build a pathname to the component we want to use to get sensor data.
sprintf(pathName,’’%s/Sonar/Segment’’,’’Robot0’’); // Robot0 is the robot server.

// Locate the component we want.
ptempObj = pHelper->find_object(pathName);

// Request the interface we want from the object we found
try
{
        pSonarSeg = MobilityGeometry::SegmentState::_narrow(ptempObj);
}
catch (...)
{
        return -1; // We’re through if we can’t use sensors.
}

// Build pathname to the component we want to use to drive the robot.
sprintf(pathName,’’%s/Drive/Command’’,’’Robot0’’); // Robot0 is the robot server.

// Locate object within robot.
ptempObj = pHelper->find_object(pathName);

// We’ll send two axes of command. Axis[0] == translate, Axis[1] == rotate.
// Positive numbers mean move forward/turn left
// Negative numbers mean move backward/turn right
OurCommand.velocity.length(2);
```

```
// Request the interface we need from the object we found.
try
{
        pDriveCommand = MobilityActuator::ActuatorState::_duplicate(
                MobilityActuator::ActuatorState::_narrow(ptempObj));
}

// Catch any errors that may occur
catch (...)
{
return -1;
}
```

After the data structures and hardware hooks are set up, one just uses a series of function calls to acquire data or manipulate the robot. Some examples are given below:

```
// Turn right for duration seconds at speed TURN_SPEED. Then stop. Remember, negative
// speeds indicate rightward movement, while positive indicate leftward movement for
// velocity[1].  The same applies for velocity[0], where positive refers to forward motion
// and negative to backwards motion. This is all done in the main thread of the program
// since current constraints do not allow one to write to the velocity data structures outside
// of the main method.
OurCommand.velocity[0] = 0.0;
OurCommand.velocity[1] = -TURN_SPEED;
pDriveCommand->new_sample(OurCommand,0);
omni_thread::sleep(duration);
OurCommand.velocity[0] = 0.0;
OurCommand.velocity[1] = 0.0;
pDriveCommand->new_sample(OurCommand,0);

// Acquire sonar data and calculate the distance from the front sonar sensor
// to the nearest object infront of it
pSegData = pSonarSeg->get_sample(0);

// Index 0 is the front sonar sensor
float tempdist = sqrt(
(pSegData->org[0].x - pSegData->end[0].x)*
(pSegData->org[0].x - pSegData->end[0].x)+
(pSegData->org[0].y - pSegData->end[0].y)*
(pSegData->org[0].y - pSegData->end[0].y));
```

6.2.2 MSocket: Working with Linux sockets in C++ (PMLb and PMLc)

The other part of the low level robot programming is the actual communication from the upper level to the lower level through PMLb. In our robot this is done in a client/server environment between the PMLb and the PMLc level using sockets. Two different languages are being used for our socket communication (C++ and LISP). We will discuss the server/C++ side first, which is part of PMLc. For the implementation of server side PMLc we chose to use the MSocket class [12] and a simple parser to parse the input from these sockets. The MSocket has a variety of functions for creating sockets, sending/receiving messages over sockets, and closing the connection and are too numerous to list, but here are the ones we used:

```
// The following are data structures and methods used in socket communication in C++
// To access the methods one includes msocket.h and makes the appropriate function calls
// in whatever program they are using socket communication in. In our case this is
// cserver.c++. This is not a full program, just the pertinent function calls and descriptions.

// Empty MSocket objects
MSocket S, C;

// Initialize socket on this server, on port thePort
S.Server(thePort);

// Accept client’s connection, waits until it gets a connection
S.Accept(C);

// Close server socket
S.Close();

// read from client socket into string buffer
C >> Buffer;

// write to the client socket the string in Buffer
C << Buffer;

// Close client
C.Close();
```

After the command is read in from the socket it is parsed to see which action should be executed by the robot. This is done as follows:

```
char *command; // command to execute
int duration; // duration to run the command

// Parse the command
command = strtok(Buffer, " ");

// Check what the command is, and parse it if necessary.
// Then execute the appropriate action
if(strcmp(command, "quit")==0)
{
        cout << ‘‘Quit the program.’’ << endl;
        break;
}
else if(strcmp(command, "forward")==0)
{
        // Run code necessary for parsing time data, and then move the robot forward
}
// Similar code is used for moving backwards, turning right, and turning left
```

6.2.3 Primitive actions and socket communication (PMLa/PMLb)

For our SNePS network to work it needs to have the capability to communicate with the primitive functions implemented in the PMLc, this is done first by writing the primitive functions in the PMLa and using the PMLb to communicate these actions to the robot using socket connections. Socket connections in C++ have already been described, and Santore [6] already has a paper on how to do multiprocessing and socket connection in LISP, so we will just show a few of the PMLa/PMLb primitive functions we are using.

```
;; Client that interfaces with the robot.

(defvar *interface-socket* nil "The socket that connects to the robot.")
(defvar *isFrontHi* nil "If the front sensors are hi (t) or low (nil).")
(defvar *isLeftHi* nil "If the left sensors are hi (t) or low (nil).")
 (if (not (find-package :mp))
     (require :process))

(defun ProcessSensorInput ()
"Listen for any messages from the server in a separate process, and take action"
        (let ((InString "")
             (socket-output-lock (mp:make-process-lock)))
                ;;while we don't read a hangup, loop and play with the input
                (loop do
                        ;;get the next input sentence - trim off nulls
                        ;;from the beginning and the end (reading to the eol
                        ;;will leave behind a null)
                        (mp:wait-for-input-available *interface-socket*)
                        (setf InString (string-trim '(#\null)
                                (read-line *interface-socket*)))
                        ;; Determine the command sent, and set the appropriate variable
                        (cond
                                ((string= InString "front low") (setf *isFrontHi* nil))
                                ((string= InString "front hi") (setf *isFrontHi* t))
                                ((string= InString "left low") (setf *isLeftHi* nil))
                                ((string= InString "left hi") (setf *isLeftHi* t)))
                until (equalp InString "Quit")))
        (close *interface-socket*))

(defun OpenConnection ()
"Open up a connection to the robot server"
        (let ((port '()) (machine "irobot.cse.Buffalo.EDU"))
                (pprint "What port is the server being run on?> ")
                (setf port (read))
                ;;now actually make the socket, connecting at the same time
                (setf *interface-socket* (socket:make-socket :remote-host machine
                                                             :remote-port port)))
        (mp:process-run-function "Input Thread" #'ProcessSensorInput))
```

```
(defun move-forward (theTime)
“Moves the robot forward for a specified time. There are similar functions for move
backward, turn right, and turn left.”
        (cond
                ((not (null *interface-socket*))
                (write-line (format *interface-socket* "forward ~A" theTime))
                (force-output *interface-socket*))))
```

6.2.4 Primitive functions needed

Magellan has a set of moves which can be described as primitive. They are the following which are implemented in Lisp. The primactions are called from the SNePS code.

Making the robot move
(1)*Move forward*: Magellan moves ahead in a straight line. It follows the left wall.

(2)*Move backward*: Magellan moves backwards in a straight line. In order to ensure that it does not bump into an obstacle/wall behind it, the sonar readings from behind the robot, or the bump readings have to be observed and accordingly planning has to be implemented.

(3)*Turn left*: Magellan makes a left turn from its original position. He rotates by 90 degrees to face the left direction from his original position. The sonar readings from the left of the robot have to be observed. This primitive allows the robot to go straight through a doorway and from one room to another. When the robots left sensors go low, it has encountered an open door, hence it needs to take a left turn to get into the doorway (it takes a left turn at an outer corner). Then, the robot goes straight through the doorway and turns left again. The robot is now in the adjacent room, beside the left wall.

(4)*Turn right:* Magellan turns 90 degrees to the right. When the robot encounters an inner corner, the sonar readings from the front of the robot and the left of the robot are both high. Hence, he has to take a right turn at the corner and continue following the left wall.

(5)*Stop:* Magellan stops.
If the only door in room R1 is closed, the robot should not keep circling the room. It should stop after some time. This is because being a hardware robot, there has to be a definite stopping condition which will eventually be satisfied if the goal(s) / sub goal(s) of the plan are not being reached. One way in which this can be achieved is by letting the robot be self aware. When the robot passes a closed door, it should know when at the next corner, that it missed a door because the door was closed. The logic can be programmed such that when the door is closed, and the Magellan Pro robot moves past the closed door, the robot just stops. Because until the room door opens, the robot will not get to its final goal plan.

7. Representing the environment in SNePS

7.1 Source Code in SNePS

With reference to the simple room which is being inhabited by Magellan, the following objects and relations are to be considered. This is code from the Combined_KL.SNePS which has the KL level SNePS code that we wrote.

```
;;; ====================================================
;;; KL for Shakey Reimplementation

;;; Lunarso Sutanto  & Trupti Devdas Nayak
;;; (December 2, 2003) (November 21, 2003)
;;; Based on M. Shanahan's "Reimplementing Shakey"
;;; ====================================================

(resetnet t)

;;;-----------------------------
;;; define objects and relations
;;;-----------------------------


(define agent act action
        class member
        object1 object2 object3
        lex arg1 rel arg2
        proper-name property
        room wall door corner
        the-door room
        connected-by-door
        next-corner
        inner-corner
        outer-corner
        corner-before corner-after
        sensor current
        )

;;; ---------------------------
;;; Load required files
;;; ----------------------------

;^^
; Load 'Lisp files'

;;; :ld <path of the lisp files>
(load "/home/csgrad/td23/irobot/pmla.lisp")

;^^

;;; describe the environment consisting of a door, 12 corners, 2 rooms and 8 walls
;;; -----------------------------------------------------------------------

(describe (assert member (r1 r2) class room))
(describe (assert member (c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12) class corner))
(describe (assert member d1 class door))

;;; Assert relations between room parts
;;; -----------------------------------

(describe (assert room r1 room r2 connected-by-door d1))

(describe (assert object1 c2 property outer-corner))
(describe (assert object1 c3 property outer-corner))
(describe (assert object1 c7 property outer-corner))
(describe (assert object1 c12 property outer-corner))

;;; I am including it here, so that the
;;; relationship between the corners can be observed. Rather than a corner being a
;;; property of the room, it is easier to think of corners as ‘before’ and ‘after’ other
;;; corners. This gives a clock-wise ordering of the corners in the room.

;;; Description of corner ordering
;;; ------------------------------------

(define between-corner1 between-corner2)

(describe (assert room r1 corner-before c1 corner-after c2))
(describe (assert room r1 corner-before c2 corner-after c3))
(describe (assert room r1 corner-before c3 corner-after c4))
(describe (assert room r1 corner-before c4 corner-after c5))
(describe (assert room r1 corner-before c5 corner-after c6))
(describe (assert room r1 corner-before c6 corner-after c1))

(describe (assert room r2 corner-before c7 corner-after c8))
(describe (assert room r2 corner-before c8 corner-after c9))
(describe (assert room r2 corner-before c9 corner-after c10))
(describe (assert room r2 corner-before c10 corner-after c11))
(describe (assert room r2 corner-before c11 corner-after c12))
(describe (assert room r2 corner-before c12 corner-after c7))

(describe (assert the-door d1 between-corner1 c2 between-corner2 c3))
(describe (assert the-door d1 between-corner1 c7 between-corner2 c12))

(describe
        (add forall ($c1-var $c2-var $door-var)
                ant (build the-door *door-var
                        between-corner1 *c1-var
                        between-corner2 *c2-var)
                cq (build the-door *door-var
                       between-corner1 *c2-var
                       between-corner2 *c1-var)))
```

```
;;; These primactions call the Lisp functions which actually make the robot move. It
;;; is necessary to load the Lisp files before these functions are called. These
;;; primactions print out the corresponding message as the robot performs the action.

^^
(define-primaction move-forward ()
        " Moving forward."
        (format t "~&*** Moving forward now. ~%"))


(define-primaction move-backward ()
         "Moving backward."
        (format t "~&*** Moving backward now. ~%"))


(define-primaction turn-left ()
        "Turning right."
        (format t "~&*** Turning left. ~%"))


(define-primaction turn-right ()
        "Turning right."
        (format t "~&*** Turning right. ~%"))


(attach-primaction
 move-forward move-forward move-backward move-backward turn-left turn-left
 turn-right turn-right believe believe disbelieve disbelieve
 sniterate sniterate
 snsequence snsequence)

;;; sense-left-prim and sense-right-prim are functions which use the sensor values
;;; returned by the lisp functions sense-left-lisp. Based on the values returned, the
;;; next step is taken.

;;; Note that a high sensor value means that the associated sensor is CLOSE
;;; to a wall and when the sensor value is low, it means that it is AWAY from a
;;; wall.

;;; -------
;;; You figure out that doing the moving actions are complete when the
;;; associated sensors fires.
;;; Note that a high sensor value means that the associated sensor is CLOSE
;;; to a wall

^^
(define-primaction sense-front-prim ()
  ;; Let variable s be the sensor data, 0 for lo, 1 for hi
    (if (null (sense-front-lisp))
        (believe-low 'front)
```

```
      (believe-high 'front)
      )

(define-primaction sense-left-prim ()
  ;; Let variable s be the sensor data, 0 for low, 1 for high
    (if (null (sense-left-lisp))
        (believe-low 'left)
      (believe-high 'left))

;;; ----------------
;;; Helper functions
;;; ----------------

(defun believe-high (direction)
  #!((perform (build action believe
                  object1 (build sensor ~direction current high)))))

(defun believe-low (direction)
  #!((perform (build action snsequence
                  object1 (build action disbelieve
                              object1 (build sensor ~direction
                                          current high))
                  object2 (build action believe
                              object1 (build min 0 max 0
                                          arg (build sensor ~direction
                                                 current high)))))))

;;; debug
(defun sense-front-lisp ()  5)

(attach-primaction sense-front-prim sense-front-prim
                sense-left-prim sense-left-prim)

^^
;;; ========================================
;;; COMPLEX ACTS

;;; follow-wall
;;; -----------
;;; Follow wall is 'self-aware', which means that in this KL level, it
;;; will know whether it's already close to a corner.
;;;
;;; Currently, this version of follow-wall will only work correctly when
;;; it's following a wall towards an inner corner. We will have to add an
;;; 'or' clause to the sensor checking (remember that when moving towards an
;;; inner corner, the front sensor is expected to go high; but when moving
;;; towards an outer corner, the left sensor is expected to go from low to
;;; high).
;;;
;;; Following a wall has these Preconditions:
```

```
;;; 1. variables *corner and *room are of the proper classes
;;; 2. the next-corner is defined
;;; 3. the wall that the robot is next to is defined
;;; 4. *not yet* Cassie expects the next-corner to be an inner or outer
;;; corner
;;;
;;; The Plan is to keep on moving forward until (FrontSensor <-> Left Sensor).
;;; If FrontSensor/LeftSensor is high, then Cassie knows it is at an inner
;;; corner. If FrontSensor/LeftSensor is low, then Cassie knows it is at
;;; an outer corner. Right now, we're simplifying this to only checking the
;;; front sensor.
;;;
;;; While the associated sensor is not believed to be High,
;;; perform primitive action move-forward
;;; and update associated sensor status
;;; Once completed, disbelieve that robot is beside-wall
;;; and believe that robot is at-corner *next-corner
;;;
;;; The associated sensor = front sensor if *next-corner is an inner corner
;;; and associated sensor = left sensor if *next-corner is an outer corner

(describe
        (assert forall ($c1 $c2 $room)
                &ant   ((build member *c1 class corner)
                        (build member *c2 class corner)
                        (build member *room class room)
                        (build room *room corner-before *c1 corner-after *c2)
                        (build beside-wall (build corner1 *c1 corner2 *c2))) ;redundant?
                cq (build act (build action follow-wall)
                        plan(build action sniterate
                                object1 ((build condition
                                        (build min 0 max 0
                                                arg (build sensor front current high))
                                                then
                                        (build action snsequence
                                        object1 (build action move-forward-prim)
                                        object2 (build action sense-front-prim)))
                                        (build else
                                        (build action snsequence
                                        object1 (build action disbelieve
                                                object1 (build beside-wall
                                                        (build corner1 *c1 corner2 *c2)))
                                                object2 (build action believe
                                                        object1 (build at-corner *c2))))))))))
;;; turn-right
;;; ----------
;;; This act will make the robot turn 90 degrees. Because of current
;;; time constraints, We have not made the turning "self-aware" as we did
;;; for follow-wall. Thus this turn-right function assumes that the robot
;;; always turn right correctly.
;;;
;;;
;;; Preconditions:
;;; 1. variables *corner and *room are of the proper classes
;;; 2. Cassie knows it's currently at an outer corner
;;; 3. Cassie knows that the corner it's at and the next corner define
;;; the door that it want to turn towards.

(describe
        (assert forall ($c1 $c2 $door $room)
                &ant ((build member *c1 class corner)
                        (build member *c2 class corner)
                        (build member *door class door)
                        (build member *room class room)
                        (build at-corner *corner1)
                        (build room *room corner-before *c1 corner-after *c2)
                        (build the-door *door between-corner1 *c1 between-corner2 *c2)

                cq ((build act (build action sturn-right)
                           plan(build action turn-right-prim))
                                (build action disbelieve
                                        object1 (build at-corner *c1))
                                (build action believe
                                        object1 (build in-doorway *door room *room1))))))


;;; turn-left - when in doorway
;;; ----------
;;; Similar to turn-right, this function is currently *not* self-aware;
;;; it assumes that the PML levels make the robot turn 90 degrees to the
;;; right and completes the action correctly.
;;;
;;; This is an snsequence of these actions in order:
;;; 1. turn left 90 degrees
;;; 2. move-forward 4 times
;;; 3. turn left 90 degrees

(describe
        (assert forall ($c1 $c2 $c3 $door $room1 $room2)
                &ant ((build member *c1 class corner)
                        (build member *c2 class corner)
                        (build member *c3 class corner)
                        (build member *door class door)
                        (build member *room1 class room)
                        (build member *room2 class room)
                        (build in-doorway *door room *room1)
                        (build the-door *door between-corner1 *c1 between-corner2 *c2)
                        (build room *room1 room *room2 connected-by-door *door)
                        (build room *room2 corner-before *c2 corner-after *c3))
cq ((build act (build action sturn-left)
            plan(build action snsequence
                  object1 (build action turn-left-prim)
                  object2 (build action move-forward-prim)
                  object3 (build action move-forward-prim)
                  object4 (build action move-forward-prim)
                  object5 (build action move-forward-prim)
                  object6 (build action turn-left-prim)))

                  (build action disbelieve
                           object1 (build in-doorway *door room *room1))
                  (build action believe
                           object1 (build beside-wall
                                             (build corner1 c2 corner2 c3))))))
```

The comments are self-explanatory, though we hope to include diagrams of their logic in a revised paper. The format of these complex actions is basically a bunch of antecedents describing the preconditions of performing that act, and then what happens when the act is performed. The act is described as a plan consisting of a sequence of primitive actions and then updates of the fluents. Do take a close look at Follow-Wall and note how the act is performed as a loop of sensor-detection and forward movement. This is actually quite an awful way of performing the action and is discussed further in the next section. Note also that there are two turn-lefts: one is for turning into a doorway and the other is for turning out of a doorway. This "polymorphism", so to speak, is possible since the preconditions to the actions are dfferent.

Robot Planning

The robot's planning is actually a simple snsequence of follow-wall and turn actions, depending on where the robot starts. For example, given that the robot starts at the wall in between corners c1 and c2, its plan will be to follow-wall, turn-left, follow-wall, turn-left, follow-wall, turn-left twice and be at the goal room. This simplistic goal-plan is sufficient for this stage.

```
;;; The robot needs to be in an initial start state prior to taking input through the
;;; sensor and changing the beliefs.

;;; Initially, front sensor not high, so it believes it is not in front of a wall
(perform (build action believe
            object1 (build min 0 max 0
                        arg (build sensor front current high))))

;;; Initially, left sensor is high, so it believes it is beside a wall
(perform (build action believe
            object1 (build sensor left current high)))

(perform (build action believe
            object1 (build beside-wall
                        (build corner1 c9 corner2 c10))))
```

7.2 Problems Encountered

A few problems were encountered during the implementation phase:

1. The original Shakey design does not take into account the fact that the robot might go off course if it moves away from the wall it is following. This is a likely event in the case where the robot wheels are not perfectly aligned and there is a chance of the path the robot is following will be skewed. In this case, the solution would be to have a cut-off for the sensor input from the left sensors, which would indicate if the robot was moving too far away from the wall. If the sensor input went low, below this threshold, it will indicate that the robot is not very near the wall.

2. Another major problem that came up was one which has been encountered on previous occasions during the implementation of other cognitive entities. The fact that in software, instructions are executed in very less time, whereas in hardware, in the read world, it takes a non-negligible amount of time to actually execute the command issued.

   The commands which were issued by the client program (the process on the Unix machine) were sent through the socket to the server process running on the robot. For example, the move-forward command. The server process on the robot is supposed to issue to command to make the hardware robot move-forward and the sensor input will be recorded after the move has been completed, and based on the new sensor input, the client on the Unix side can issue the next command. The problem being faced was that the client process was in a loop where it would send a move-forward command, and check for changes in sensor input and issue a move-forward command again. The looping was too fast for the real world implementation to complete executing the move-forward commands. Too many move-forward commands were queuing up on the server side (robot) because of which the actions performed were not what was expected.

   A few solutions were suggested. One of which was to take care of the changes in sensor input on the server side itself. (robot side). Hence, when a move-forward command is issued, the robot moves forward until a change in sensor input, which will indicate it's at a corner or beside an open door. Then the client SNePS program can follow appropriate logic and issue the next command. We are working on implementing this.

3. Mobility data structures can only be accessed from the main thread it seems.

4. The robot is currently tethered to the wall by an ethernet cable, which limits the mobility of the robot.

5. Learn X11 libraries in C++ to work with the camera.

8 Integration

The final step of the project is the integration of all the modules each participant of the seminar has produced. One is the PML level code. Another is the KL SNePS representation of environment and rules. These should be integrated for the robot to use its knowledge base (KL) to determine a goal and subsequently build a plan to reach that goal. During the course of executing that plan, it also uses the Lisp functions which are linked to primactions through SNePS. The robot is goal-oriented and keeps planning and replanning till it reaches its goal, or its goal is deemed unreachable, at which point it stops.

During integration of the various modules, there was not much difficulty. The SNePS functions call the Lisp functions and the variables for sensor input are accessible to SNePS.

9. Future enhancements

There is immense scope for future implementations being more ambitious because of the wide range of capabilities possessed by Magellan. Some of the future aims are

1. The initial aim was the more ambitious implementation of a *delivery agent*. Given a packet and a destination where the packet has to be delivered, Magellan uses prior knowledge, and mapping techniques to chart out a path from his current position to the destination. The proposed use for the onboard camera was identification of people based on the color of their clothes. Every person will be identified by a blob of color on top (shirt) and a blob of color on the bottom (jeans).
   Magellan will reach the person to whom the packet has to be delivered, using the mapping techniques and knowledge of the floor(s), (he might even use the elevator) .The mapping techniques are to be chosen from either topological mapping or grid based mapping or a combinational algorithm which makes use of both concepts.

2. Learn X11 libraries in C++ to work with the camera.

3. Develop follow wall primitive instead of just having move-forward, move-backward, move-left and move-right.

9.2 Ideas for implementing a wider range of tasks

Because the hardware robot provides an excellent way of hands on programming, it is important to list the different hardware features which can be used to enhance the capabilities of the robot.

8.2.1 Camera
Identify people through blobs of color

8.2.2. Mapping
Robust Topological mapping (explore and learn)

8.2.3. Speech driven robot.
A sound card has to be fitted onto the robot, so that it can respond to natural language commands along with command strings through the Unix client.

10. Summary

There are still many obstacles to overcome with these new systems we are working with, despite the many that have been overcome already. This has been an extremely good learning experience. We hope to successfully implement all the proposed ideas. *Magellan is the reinvented Shakey*.

11. References/Bibliography

[1] Stuart C. Shapiro and The SNePS Implementation Group. SNePS 2.6 User's Manual Department of Computer Science and Engineering, University at Buffalo, The State Universtiy of New York, Buffalo, NY, October 7, 2002

[2] Scott Napieralski. *Dictionary of CVA SNePS Case Frames*
http://www.cse.buffalo.edu/~stn2/cva/case-frames/

[3] Stuart C. Shapiro and Haythem O. Ismail. *Anchoring in a Grounded Layered Architecture with Integrated Reasoning.* Robotics and Autonomous systems 43 (2003) 97-108.

[4] Stuart C. Shapiro, The SNePS Approach to Cognitive Robotics, presented to LIMSI-CNRS, Universite de Paris Sud, Orsay, France, December 19, 2002. (A powerpoint presentation)

[5] M.P.Shanahan. Reinventing Shakey, in Logic-Based Artificial Intelligence.
ed. Jack Minker, Kluwer Academic. 2000
pages 233-253.

[6] John F. Santore. Multiprocessing, Semaphores and Networking with ACL, SNeRG Technical Note 33, Department of Computer Science and Engineering, University at Buffalo, The State Universtiy of New York, Buffalo, NY, September 5, 2002

[7] M.P.Shanahan, The Event Calculus Explained, in *Artificial Intelligence Today*, ed. M.J.Wooldridge and M.Veloso, Springer Lecture Notes in Artificial Intelligence no. 1600, Springer (1999), pages 409-430.

[8] IRobot.
http://www.irobot.com/

[9] IRobot’s Mobility Manual
http://www.cse.buffalo.edu/shapiro/Courses/CSE716/MobilityManRev4.pdf

[11] N.J.Nilsson, ed. Shakey the Robot, SRI Technical Note no. 323 (1984), Menlo Park, California

[12] MSocket Class.
http://www.code.ghter.com/linux2.html