# Undergraduate Report

## REU Report: Designing a Universal Robotics Platform

*by Stephanie Wojtkowski*
*Advisor: P.S. Krishnaprasad*

**U.G. 99-1**

*ISR develops, applies and teaches advanced methodologies of design and analysis to solve complex, hierarchical, heterogeneous and dynamic problems of engineering technology and systems for industry and government.*

*ISR is a permanent institute of the University of Maryland, within the Glenn L. Martin Institute of Technology/A. James Clark School of Engineering. It is a National Science Foundation Engineering Research Center.*

***Web site http://www.isr.umd.edu***

**Designing a Universal Robotics Platform**

Written by:

Stephanie Wojtkowski
Research Experience for Undergraduates (REU)
Institute for Systems Research(ISR)
University of Maryland, College Park

8/6/1999

Dr. P.S. Krishnaprasad
Intelligent Servosystems Laboratory(ISL)
University of Maryland
College Park, MD 20742

Stephanie Wojtkowski
August 6, 1999

# Designing a Universal Robotics Platform

## 1. Introduction

Computer software is generally designed to operate in a very specific environment. In contrast, the diverse demands of robotics necessitate the design of a system that is generic enough to operate on any robot. In 1990, Dr. Roger Brockett proposed the creation of Motion Description Language (MDL), a device independent language that would integrate the simplicity of behavior-based programming and the complex mathematics associated with control theory. His colleague, Dr. P.S. Krishnaprasad, expanded on this idea, formulating Motion Description Language extended, or MDLe. In a 1998 research paper, Dr. Krishnaprasad and two associates outline a framework that breaks down the actions of a robot into smaller and smaller pieces until reaching the level that directly governs the speeds of the robot's wheels. These sets of wheel controls can be grouped together to form atoms, simple motions like moving forward or stopping. These atoms can then be grouped into behaviors, such as following a wall. Finally, these behaviors can form plans, like finding a specified lab or delivering mail. A planner system will then use these discrete actions and the principles of control theory to successfully navigate the robot through an unknown environment. Such a broad based platform would be ideal for various robotics applications.

## 2. Background

The robot used for this project was a Nomad Super Scout II manufactured by Nomadic Technologies. This model is equipped with 16 sonars and bumpers to collect sensory data. The actual robot looks like this (Figure 1):

**Figure 1**. Nomad Super Scout II, Nomadic Technologies.

Ideally, robots would have perfect mobility, with the ability to roam without directional constraints. However, most robots, including our research subject, are driven by wheels, which apply a non-holonomic constraint to the robot. In essence, the robot cannot directly move perpendicular to the wheels. The simplest example of this phenomenon is a unicycle, a single wheel that can move forward, move backward, and turn. The motion of such a machine is modeled by:

$$\dot{G}=G(A_1 u_1+A_2 u_2)$$

where G, $A_1$, $A_2 \in$ SE(2) are given by:

$$G=\begin{matrix}\cos\theta & -\sin\theta & x \\ \sin\theta & \cos\theta & y \\ 0 & 0 & 1\end{matrix}, \quad A_1=\begin{matrix}0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0\end{matrix}, \quad A_2=\begin{matrix}0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0\end{matrix}$$

and the angular velocity, v1, and translational velocity, v2, are given by:

$$v_1=\frac{u_L-u_R}{W} \quad v_2=\frac{u_L+u_R}{2}$$

If the desired position for the unicycle is one parallel to its current position, the unicycle must adopt a motion similar to that of parallel parking a car. This limitation must be considered when modeling the robot.

Matlab models of this motion proved to be very useful in understanding the dynamics of this system. Euler's method approximates the integral of the differential equation above at any instant of time. The following equation gives the position of the robot using Euler's method:

$$x_T=x_{T-1}(A_1 v_1+A_2 v_2)\Delta T+x_{T-1}$$

As the time interval ($\Delta$T) approaches zero, the approximation becomes more and more accurate. The graph shown below (Figure 2) illustrates the motion of a unicycle using this equation given rotatonal and translational inputs of 1.

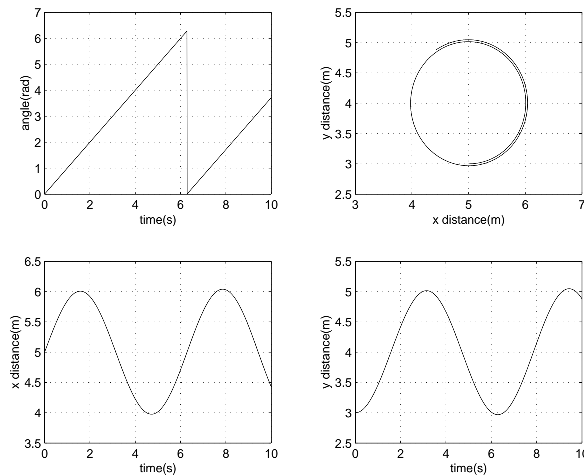


**Figure 2**. Matlab model of differential equation describing non-holonomic constraints. Inputs for rotational and translational velocities are 1.

Various inputs can be explored through this modeling tool. The following graphs were created using a delta function for the two inputs (Figure 3).

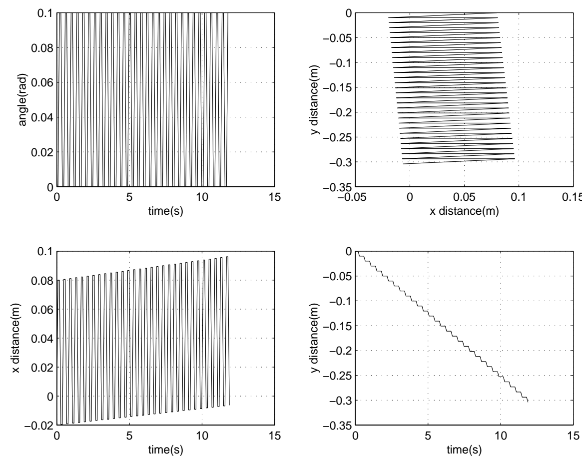


**Figure 3**. Matlab model of differential equation given a delta function as inputs for rotational and translational velocity. Known as 'parallel parking' motion.

With models like the ones shown, we can predict the path of our robot without actually testing controls on the robot. The negative aspect to theoretical testing is that it disregards the complications that are experienced in real systems. In reality, the

wheels of the robot may slip, causing the position of the robot to be imprecise. The time necessary to send commands from the software to the robot and back is also not considered. These are just a few of the discrepancies that could cause the robot to act differently from the simulated model.

## 3. MDLe System

### 3.1. Basic Subsystems

The concerns mentioned in Section 2 are purely restrictions specific to our robot. The methodology of MDLe, however, follows a device independent format. Theoretically, this language could be used to communicate with a submersible or a satellite with equal ease. This platform will be composed of 6 main subsystems: the language interpreter, the motion planner, the cartographer, the scheduler, the knowledge base and the virtual robot. The language interpreter converts a command file provided by the user into an executable file in a format that the robot can understand. A desired task is received by the motion planner, which then creates a plan to achieve this task. This module would make the robot autonomous. The cartographer is a module that maps the robot's surroundings, a useful tool for navigation purposes. The scheduler contains two parts, the timer and the programmable interrupt interface, or PII. The timer executes each set of controls at the right instant, while the PII handles any interrupts that occur at run time. The collection of all atoms available for constructing plans is known as the knowledge base. The system will eventually be capable of selecting actions from this base to execute a specified task. Finally, the virtual robot is an interface between the MDLe system and the real robot. The diagram below shows a schematic representation of this model (Figure 4):

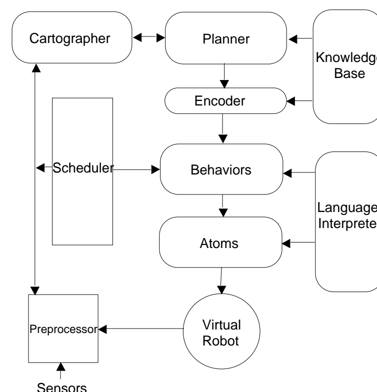


**Figure 4.** Schematic drawing of the MDLe system.

### 3.2. Creating Plans

MDLe is designed with an interrupt-based methodology. This means that for

each level of motion there are predefined conditions that signal the end of a motion. These conditions are tested periodically and, if one of the conditions is true, the action is terminated. For example, the robot may stop going forward when it hits a wall. In this case, hitting the wall is an interrupt. The MDLe platform is designed with this usage in mind.

The most basic level of controls contains the inputs for the speeds of the two wheels. The lowest layer in the planner, then, is a Control class, which allows the user to set the speeds of the two wheels. The number of controls for the robot can be specified to conform to a specific robot. In our case, the robot needed two controls, one for the left wheel ($u_L$) and one for the right wheel ($u_R$). More sophisticated machines may also have a steering wheel or a third degree of freedom. In these cases, the length of the array of controls is simply set to a different value.

A collection of controls is known as an atom, a simple motion often described by only one set of controls. The Atom class contains a vector that holds this series of controls. A second vector runs parallel to it and contains the period over which each control is to be executed. Each atom is also equipped with an execute function that runs each set of controls in order. To adhere to the interrupt-based methodology, the Atom class is equipped with a default interrupt. This function contains a bumper check, which makes sure that the robot has not hit anything. Under construction is a method that will run the controls in reverse, making the robot retrace the path it just completed.

A behavior consists of a list of atoms to be executed in the specified order. At this level, it is possible to loop through the atoms ad infinitum. Suppose, for example, that the user wants the robot to continue down a hallway, correcting to stay in range of the wall, until it reaches a corner. The robot should loop through an atom that moves it forward followed by an atom that corrects its distance from the wall indefinitely. The occurrence of reaching a corner will then stop the behavior, no matter what stage it is in.

Plans are constructed in the same manner as behaviors. They have lists of behaviors that perform the same tasks as the list of atoms in a behavior.

Based on this hierarchical system, intricate plans like mapping building can be broken down into less intimidating pieces.

### 3.3. Data Structures

An array was chosen to hold the wheel controls for the robot due to its simple manipulation. A dynamic data structure was unnecessary because the number of controls for a specific robot is declared at the beginning of the class and is constant. Controls will also never need to be removed, an action that occurs in linear time for an array.

The set of controls in the atom class required a more flexible data structure, a vector. An atom can consist of any number of controls, so a dynamic structure like a vector is necessary to allow for expansion. A vector is sufficient because insertions and deletions are only performed at the end of the vector, an operation that occurs in constant time. If controls were added to or deleted from the beginning, the operation would be executed in linear time, making it less efficient than other structures.

The Behavior class uses a singly-linked list to store its atoms. Similarly, the Plan class employs a singly-linked list to hold behaviors. This type of list allows for constant time insertion and removal at both ends, making addition and deletion from anywhere in the list more efficient. Unfortunately, there is no way to run the behavior or plan in reverse order since singly-linked lists are only equipped for forward iteration. The singly-linked list will have to be converted to a doubly-linked list, which allows for both forward and reverse iteration, if this capability becomes necessary.

### 3.4. Sample Program

To test the planner of the MDLe system, a sample program was written that has the robot follow a moving object. Due to time constraints, the project was simplified to operate only in an environment with no obstacles. The moving object must also remain within range of the robot's sonars. The program relies on a function from the cartographer that returns the coordinates from which an object has moved. The robot will then move to these coordinates and wait for another motion. This algorithm has the robot following one step behind the moving object, a discrepancy that I hope will be resolved in real time execution.

The task was divided into four atoms intended to loop continuously: Wait, Orient, GoTo, and Stop. Wait does not move the robot at all; the robot simply waits until it receives data signaling that something has moved. The Orient class then turns the robot in the direction of the coordinates from which an object has moved. GoTo moves the robot toward those coordinates, and Stop stops it when it arrives at the proper position. This is an example of a behavior that would run ad infinitum, allowing the robot to follow a moving object continuously. This program is still in the debugging stage but is nearly complete.

## Future Research

The goal of this research is creating a planner that can navigate a completely unknown environment autonomously to reach a desired goal. This portion of the project will rely heavily on control theory. The differential equation describing the motion of the robot will be solved to provide all possible trajectories. Some will be eliminated by the non-holonomic constraint, and the rest will be considered to find the most efficient path.

In addition, we will continue to debug the platform that we have created to make it more generally useful. Since efficiency is essential when designing a system to operate in real time, this platform will be revised to operate as quickly as possible.

## Acknowledgements

I would like to extend thanks to Dr. Krishnaprasad for putting together this project. I would also like to thank Sean Andersson and Fumin Zhang. They did a wonderful job of explaining the project to us and incorporating us into their graduate research. Ram Venkataraman introduced us to the mathematics that are central to mobile robotics, along with introducing us to the power of Matlab. Diane Ihasz worked

hard to make sure that the program was interesting and educational.

## Appendix 1

```
/****************************************************************/
/* Control.h                                                    */
/*--------------------------------------------------------------*/
/*     class that acts as an interface for all MDLe controls    */
/*                                                              */
/*                                                              */
/*   Created by:Stephanie Wojtkowski                            */
/*   Created on:6-24-1999                                       */
/*   Last Modified: 7-08-1999                                   */
/****************************************************************/
#ifndef __CONTROL__
#define __CONTROL__

#include "GRobot.h"

#define DIMENSION 2                       /*The number of controls
                                          necessary for
                                          this particular robot*/

class Control{
private:
  float U[DIMENSION];                     /*An array of controls for
the robot*/

public:
  Control();                              /*Constructor*/
  Control(float z[DIMENSION]);            /*Alternative
constructor*/
  float GetUi(int n);                     /*Return a specific
control*/
  int SetUi(int n, float z);              /*Set a specific control*/
  int GetDimension();                     /*Get the number of
controls*/
  void SetControls(float z[DIMENSION]);/*Set all controls*/
};
#endif
```

```
/**************************************************************/
/* Control.cpp                                                */
/*------------------------------------------------------------*/
/*                 member functions for class Control         */
/*                                                            */
/*                                                            */
/*   Created by:Stephanie Wojtkowski                          */
/*   Created on:6-28-1999                                     */
/*   Last Modified: 7-08-1999                                 */
/**************************************************************/

#include "Control.h"

/*Constructor*/
Control::Control(){
  for(int i = 0; i<DIMENSION; i++)
    U[i] = 0;
}

/*Alternative constructor*/
Control::Control(float z[DIMENSION]){
  for(int i = 0; i<DIMENSION; i++)
    U[i] = z[i];
}

/*Set a specific control*/
int Control::SetUi(int n, float z){
  if(n<0 || n>DIMENSION)
    return FAILURE;
  U[n] = z;
  return SUCCESS;
}

/*Return a specific control*/
float Control::GetUi(int n){
  return(U[n]);
}

/*Set all controls*/
void Control::SetControls(float z[DIMENSION]){
    for(int i = 0; i<DIMENSION; i++)
      U[i] = z[i];
}

/*Get the number of controls*/
int Control::GetDimension(){
  return DIMENSION;
}
```

```
/*************************************************************/
/* Atombase.h                                                */
/*-----------------------------------------------------------*/
/*       class that acts as an interface for all MDLe atoms  */
/*                                                           */
/*                                                           */
/*   Created by:Stephanie Wojtkowski                         */
/*   Created on:6-24-1999                                    */
/*   Last Modified: 7-08-1999                                */
/*************************************************************/
#ifndef __ATOMBASE__
#define __ATOMBASE__

#include "Control.h"
#include "stlinclude.h"
#include "GRobot.h"

class Atombase{
 private:
  vector <Control> U;           /*The controls at any time T*/
  vector <float> T;                 /*Time intervals that
                                      correspond with U*/
  Control scaling;                  /*Scaling Parameters*/
  CRobot *robot;                    /*A pointer to a robot*/
  int size;                         /*The number of controls in
                                      U*/
  float timeVal;                    /*The number of seconds in
                                      this atom*/
  float timeScale;                  /*The time scaling
                                      parameter*/
  int callCount;                    /*Number of times this
                                      control is called*/

public:
  Atombase(CRobot *);                        /*Constructor*/
  CRobot *GetRobot();                     /*Get a pointer to the
                                            robot*/
  virtual int Execute();                     /*Run each set of
                                               controls*/
  virtual int Interrupt();                   /*Interrupt the
                                               current atom*/
  virtual float Stop();                      /*Nothing; will be a
                                               stop function*/
  virtual void SetU(vector<Control> v);      /*Set the vector of
                                               controls*/
  virtual void SetUt(Control z, int t);      /*Set a specific set
                                               of controls*/
  virtual Control GetUt(int t);              /*Get a specific U*/
  virtual int SetScaling(Control c);         /*Set scaling
                                               parameters*/
  virtual Control GetScaling();              /*Get scaling
                                               parameters*/
```

```
  virtual int Reverse();                  /*Run the controls in
                                            reverse*/
  virtual void SetTimeVal(float n);       /*Set time value*/
  virtual void SetTimeVal(float t, float s);/*Set timeVal and
                                             timeScale*/
  virtual float GetTimeVal();             /*Get time value*/
  virtual void SetTimeScale(float n);     /*Set time scale*/
  virtual float GetTimeScale();           /*Get time scale*/
  virtual void AddControl(Control c, float n);/*Add a control to
                                               the end*/
  virtual void DeleteControl(int n);      /*Delete control at
                                            specified index*/
};

#endif
```

```
/************************************************************/
/* Atombase.cpp                                             */
/*----------------------------------------------------------*/
/*      member functions for class Atombase                 */
/*                                                          */
/*                                                          */
/*   Created by:Stephanie Wojtkowski                        */
/*   Created on:6-24-1999                                   */
/*   Last Modified: 7-08-1999                               */
/************************************************************/

#include "Atombase.h"
#include <iostream.h>

int i = 0;

/*Constructor*/
Atombase::Atombase(CRobot * r){
  int i;
  float z[] = {0, 0};

  size = 1;
  robot = r;
  callCount = 0;
  Control temp(z);

  /* Initialize U, and T so that they contain at least one
component)*/
  U.insert(U.begin(),z);
  T.insert(T.begin(),1);

  //for(i = 0; i<size; i++){
  //U[i].SetControls(z);
  //
  for(i = 0; i<scaling.GetDimension(); i++){
    scaling.SetUi(i, 1);
  }
  timeVal = timeScale = 1;
}

/*Get a pointer to the robot*/
CRobot *Atombase::GetRobot(){
  return robot;
}

/*Run each set of controls*/
int Atombase::Execute(){
  static int i = 0;

  if(i >= this->size)
    return FAILURE;
  else{
```

```
    if(callCount*DEFAULT_TIMER<T.operator[](i)){
      (*robot).Drive(
(U.operator[](i)).GetUi(0)*scaling.GetUi(0),
                (U.operator[](i)).GetUi(1)*scaling.GetUi(1) );
      callCount++;
    }
    else{
      i++;
      if(i>=this->size)
     return FAILURE;
      (*robot).Drive(
(U.operator[](i)).GetUi(0)*scaling.GetUi(0),
                (U.operator[](i)).GetUi(1)*scaling.GetUi(1) );
      callCount = 1;
    }
   return SUCCESS;
  }
}

/*Interrupt the current atom*/
int Atombase::Interrupt(){
  if(State[33])
    return TRUE;
  return FALSE;
  //return Bumpercheck();
}

/*Nothing; will be a stop function*/
float Atombase::Stop(){
  return 0;
}

/*Set the vector of controls*/
void Atombase::SetU(vector<Control> v){
  for(int i = 0; i<size; i++){
    U[i] = v[i];
  }
}

/*Set a specific set of controls*/
void Atombase::SetUt(Control z, int t){
  U[t] = z;
}

/*Get a specific U*/
Control Atombase::GetUt(int t){
  return (U.operator[](t));
}

/*Set scaling parameters*/
int Atombase::SetScaling(Control c){
  int i;
```

```
  if(c.GetDimension() != U[0].GetDimension())
    return FAILURE;

  scaling = c;
  return SUCCESS;
}

/*Get scaling parameters*/
Control Atombase::GetScaling(){
  return scaling;
}

/*Run the controls in reverse*/
int Atombase::Reverse(){
  static int i = size;

  if(i<1)
    return FAILURE;
  else{
    i--;
    //    (*robot).DifferentialDrive( U.operator[](i));
    return SUCCESS;
  }
}

/*Set time value : overloaded*/
void Atombase::SetTimeVal(float n){
  timeVal = n;
}

/*Set time value and time scaling : overloaded*/
void Atombase::SetTimeVal(float t, float s){
  timeVal = t;
  timeScale = s;
}

/*Get time value*/
float Atombase::GetTimeVal(){
  return timeVal*timeScale;
}

/*Set time scale*/
void Atombase::SetTimeScale(float n){
  timeScale = n;
}

/*Get time scale*/
float Atombase::GetTimeScale(){
  return timeScale;
}

void Atombase::AddControl(Control c, float n){
```

```
  static int switch1 = 0;

  if(switch1 == 0){
    for(int i = 0; i<U[0].GetDimension(); i++){
      U[0].SetUi(i, c.GetUi(i));
    }
    T[0] = n;
    switch1 = 1;
  }
  else{
    U.push_back(c);
    T.push_back(n);
    size++;
  }
}

void Atombase::DeleteControl(int n){
  vector<Control>::iterator it1 = U.begin();
  vector<float>::iterator it2 = T.begin();
  for(int i = 0; i < n; i++){
    it1++;
    it2++;
  }
  U.erase(it1);
  T.erase(it2);
}
```

```
/***************************************************************/
/* Behaviorbase.h                                              */
/*-------------------------------------------------------------*/
/*      class that acts as an interface for all MDLe behaviors */
/*                                                             */
/*                                                             */
/*   Created by:Stephanie Wojtkowski                           */
/*   Created on:6-24-1999                                      */
/*   Last Modified: 7-08-1999                                  */
/***************************************************************/
#ifndef __BEHAVIORBASE__
#define __BEHAVIORBASE__

#include "Atombase.h"
#include "stlinclude.h"
#include "GRobot.h"

class Behaviorbase{
private:
  slist<Atombase *> Atomlist;   /*The list of atoms in this
                                   behavior*/
  CRobot *robot;                /*A pointer to a robot*/
  float timeVal;                /*The number of seconds in this
                                   behavior*/
  float timeScale;              /*The time scaling parameter*/
  int looping;

public:
  Behaviorbase(CRobot *);              /*Initialize the robot*/
  void AddAtom(Atombase *name);        /*Add an atom to this
                                         behavior*/
  Atombase *RemoveAtom();              /*Remove an atom from this
                                         behavior*/
  virtual Atombase * Execute();        /*Execute the current
                                         atom*/
  virtual int Interrupt();             /*Interrupt the current
                                         behavior*/
  virtual float Stop();                /*Nothing; will be a stop
                                         function*/
  //  virtual Atombase *Reverse();     /*Executes in reverse*/
  virtual void SetTimeVal(float n);    /*Set time value*/
  virtual void SetTimeVal(float t, float s);/*Set timeVal and
                                         timeScale*/
  virtual float GetTimeVal();          /*Get time value*/
  virtual void SetTimeScale(float n);  /*Set time scale*/
  virtual float GetTimeScale();        /*Get time scale*/
  CRobot *GetRobot();                  /*Get a pointer to the
                                         robot*/
  void SetLooping(int n);                   /*Set looping*/
  int GetLooping();                         /*Get looping*/
};
#endif
```

```
/************************************************************/
/* Behaviorbase.cpp                                         */
/*----------------------------------------------------------*/
/*           member functions for class Behaviorbase        */
/*                                                          */
/*                                                          */
/*   Created by:Stephanie Wojtkowski                        */
/*   Created on:6-24-1999                                   */
/*   Last Modified: 7-08-1999                               */
/************************************************************/

#include "Behaviorbase.h"

//Initialize the robot
Behaviorbase::Behaviorbase(CRobot *r){
  robot = r;
  timeVal = 0;
  timeScale = 1;
  looping = FALSE;
  }

/*Get a pointer to the robot*/
CRobot *Behaviorbase::GetRobot(){
  return robot;
}

//Add an atom to this behavior
void Behaviorbase::AddAtom(Atombase *name){
  slist<Atombase *>::iterator back =
Atomlist.previous(Atomlist.end());
  back = Atomlist.insert_after(back, name);
}

//Remove an atom from this behavior
Atombase *Behaviorbase::RemoveAtom(){
  Atombase *temp;

  if(Atomlist.size()!=0){
    temp = Atomlist.front();
    Atomlist.pop_front();
    return(temp);
  }
  return FAILURE;
}

//Execute the current atom
Atombase * Behaviorbase::Execute(){
  static slist<Atombase *>::iterator it = Atomlist.begin();
  static int first = 1;

  if(it == Atomlist.end()){/*Atomlist.end() is a null node*/
```

```
    if(looping == FALSE)
      return FAILURE;
    else{
      it = Atomlist.begin();
      first = 1;
    }
  }

  if(first == 1){
    //    (*(*it)).Execute();
    first = 0;
    return (*it);
   }
  else{
    it++;
    if(it == Atomlist.end()){/*Atomlist.end() is a null node*/
      if(looping == FALSE)
     return FAILURE;
      else{
     it = Atomlist.begin();
     first = 1;
      }
    }
    //(*(*it)).Execute();
    return(*it);
  }

  /*
    static iterator it = Atomlist.begin();

    it++;
    if(*it == Atomlist.end()){
       return FAILURE;
    }
    else{
      (*(*it)).Execute();
      return (it);
    }
    */
}

//Interrupt the current behavior
int Behaviorbase::Interrupt(){
  if(State[33])
    return TRUE;
  return FALSE;
  //  return Bumpercheck();
}

//Nothing; will be a stop function
float Behaviorbase::Stop(){
  return 0;
```

```
}


/*
//Execute in reverse; can't traverse slist backwards
Atombase * Behaviorbase::Reverse(){
  static iterator it = Atomlist.end();

  if(it == Atomlist.begin())
    return FAILURE;
  else{
    it--;
    (*(*it)).Execute();
    return(*it);
  }
}
*/

/*Set time value*/
void Behaviorbase::SetTimeVal(float n){
  timeVal = n;
}

/*Set time value and time scaling : overloaded*/
void Behaviorbase::SetTimeVal(float t, float s){
  timeVal = t;
  timeScale = s;
}

/*Get time value*/
float Behaviorbase::GetTimeVal(){
  return timeVal*timeScale;
}

/*Set time scale*/
void Behaviorbase::SetTimeScale(float n){
  timeScale = n;
}

/*Get time scale*/
float Behaviorbase::GetTimeScale(){
  return timeScale;
}

/*Set looping*/
void Behaviorbase::SetLooping(int n){
  looping = n;
}

/*Get looping*/
int Behaviorbase::GetLooping(){
  return looping;
```

```
}
```

```
/****************************************************************/
/* Planbase.h                                                   */
/*--------------------------------------------------------------*/
/*      class that acts as an interface for all MDLe plans      */
/*                                                              */
/*                                                              */
/*    Created by:Stephanie Wojtkowski                           */
/*    Created on:6-24-1999                                      */
/*    Last Modified: 7-08-1999                                  */
/****************************************************************/

#ifndef __PLANBASE__
#define __PLANBASE__

#include "Behaviorbase.h"
#include "stlinclude.h"
#include "GRobot.h"

#define NOTRUN 0
#define RUNNING 1
#define TERMINATED 2

class Planbase{
private:
  slist<Behaviorbase *> Behaviorlist;    /*The list of behaviors
                                           in this plan*/
  CRobot *robot;                         /*A pointer to a robot*/
  float timeVal;                         /*The number of seconds
                                           in this plan*/
  float timeScale;                       /*The time scaling
                                           parameter*/
  int runFlag;                         /*Plan is not run,
                                         running, terminated*/
  int looping;               /*Should the planner loop through
                               the behaviors?*/

public:
  Planbase(CRobot *);                      /*Initialize robot*/
  CRobot *GetRobot();                  /*Get a pointer to the
                                         robot*/
  void AddBehavior(Behaviorbase *name);    /*Add a behavior to
                                             this plan*/
  virtual Behaviorbase *RemoveBehavior();  /*Remove a behavior
                                             from this plan*/
  virtual Behaviorbase *Execute(Atombase ** atom);/*Execute
                                               current behavior*/
  virtual int Interrupt();                 /*Interrupt the
                                             current plan*/
  virtual float Stop();                    /*Nothing; will be a
                                             stop function*/
  //  virtual Behaviorbase *Reverse(Atombase ** atom);/*Execute
                                                   in reverse*/
```

```
  virtual void SetTimeVal(float n);        /*Set time value*/
  virtual void SetTimeVal(float t, float s);/*Set timeVal and
                                             timeScale*/
  virtual float GetTimeVal();              /*Get time value*/
  virtual void SetTimeScale(float n);      /*Set time scale*/
  virtual float GetTimeScale();            /*Get time scale*/
  void SetFlag(int n);                     /*Set the runFlag*/
  int GetFlag();                           /*Get the runFlag*/
  void SetLooping(int n);                     /*Set looping*/
  int GetLooping();                           /*Get looping*/
};
#endif
```

```
/************************************************************/
/* Planbase.cpp                                              */
/*----------------------------------------------------------*/
/*          member functions for class Planbase              */
/*                                                           */
/*                                                           */
/*    Created by:Stephanie Wojtkowski                        */
/*    Created on:6-24-1999                                   */
/*    Last Modified: 7-08-1999                               */
/************************************************************/

#include "Planbase.h"

//Initialize robot
Planbase::Planbase(CRobot *r){
  robot = r;
  timeVal =0;
  timeScale = 1;
  runFlag = NOTRUN;
  looping = FALSE;
}

/*Get a pointer to the robot*/
CRobot *Planbase::GetRobot(){
  return robot;
}

//Remove a behavior from this plan
Behaviorbase *Planbase::RemoveBehavior(){
  Behaviorbase *temp;

  if(Behaviorlist.size()!=0){
    temp = Behaviorlist.front();
    Behaviorlist.pop_front();
     return(temp);
  }
  return FAILURE;
}

//Add a behavior to this plan
void Planbase::AddBehavior(Behaviorbase *name){
  static slist<Behaviorbase *>::iterator back =
    Behaviorlist.previous(Behaviorlist.end());

  back = Behaviorlist.insert_after(back, name);
 }

//Execute the current behavior
Behaviorbase * Planbase::Execute(Atombase * * atom){
    static slist<Behaviorbase *>::iterator it =
Behaviorlist.begin();
    static int first = 1;
```

```
    /*Behaviorlist.end() is a null node*/
    if(it ==Behaviorlist.end()){
      if(looping == FALSE)
     return FAILURE;
      else{
     it = Behaviorlist.begin();
     first = 1;
      }
    }

  if(first == 1){
    *atom = (*(*it)).Execute();
    first = 0;
    return (*it);
  }
  else{
    it++;
    if(it ==Behaviorlist.end()){
      if(looping == FALSE)
     return FAILURE;
      else{
     it = Behaviorlist.begin();
     first = 1;
      }
    }
    *atom = (*(*it)).Execute();
    return(*it);
  }

  /*static iterator it = Behaviorlist.begin();
  Atombase *result;

  it++;
    result = (*(*it)).Execute();
    if(result == 0){
      it++;
     if(*it == Behaviorlist.end()){
       return FAILURE;
     }
     else{
       return SUCCESS;
     }
    }
  }*/
}

//Interrupt the current plan
int Planbase::Interrupt(){
  if(State[33])
    return TRUE;
  return FALSE;
  //return Bumpercheck();
```

```
}

//Nothing; will be a stop function
float Planbase::Stop(){
  return 0;
}


/*
//Execute in reverse; can't traverse slist backwards
Behaviorbase * Planbase::Reverse(Atombase * * atom){
    static iterator it = Behaviorlist.end();

  if(it == Behaviorlist.begin())
    return FAILURE;
  else{
    it--;
    *atom = (*(*it)).Execute();
    return(*it);
  }
}
*/

/*Set time value*/
void Planbase::SetTimeVal(float n){
  timeVal = n;
}

/*Set time value and time scaling : overloaded*/
void Planbase::SetTimeVal(float t, float s){
  timeVal = t;
  timeScale = s;
}

/*Get time value*/
float Planbase::GetTimeVal(){
  return timeVal*timeScale;
}

/*Set time scale*/
void Planbase::SetTimeScale(float n){
  timeScale = n;
}

/*Get time scale*/
float Planbase::GetTimeScale(){
  return timeScale;
}

void Planbase::SetFlag(int n){
  if( n>2 || n<0)
    runFlag = -1;       /*Set runFlag to error value*/
```

```
  else
    runFlag = n;
}

int Planbase::GetFlag(){
  return runFlag;
}

/*Set looping*/
void Planbase::SetLooping(int n){
  looping = n;
}

/*Get looping*/
int Planbase::GetLooping(){
  return looping;
}
```

**Appendix 2**

```
/****************************************************************/
/* Wait.h                                                       */
/*--------------------------------------------------------------*/
/*    class that acts as an interface for Wait functions        */
/*                                                              */
/*                                                              */
/*   Created by:Stephanie Wojtkowski                            */
/*   Created on:7-27-1999                                       */
/*   Last Modified: 7-27-1999                                   */
/****************************************************************/
#ifndef __WAIT__
#define __WAIT__

#include "GRobot.h"
#include "PlanFiles.h"
#include "CMap.h"

class Wait : public Atombase{
 private:
  Point object;      /*The position of the object to follow*/
 public:
  Wait(CRobot * r);
  int Interrupt();
  Point GetPoint();
};

#endif
```

```
/****************************************************************/
/* Wait.cpp                                                     */
/*--------------------------------------------------------------*/
/*    member functions for class Wait                           */
/*                                                              */
/*                                                              */
/*   Created by:Stephanie Wojtkowski                            */
/*   Created on:7-27-1999                                       */
/*   Last Modified: 7-27-1999                                   */
/****************************************************************/
#include "Wait.h"

Wait::Wait(CRobot *r){
  object = chasecheck();
  Atombase::Atombase(r);
  Control c(0, 0);
  Atombase::SetUt(c, 0);
  Atombase::SetTimeVal(1000);
}

int Wait::Interrupt(){
  object = chasecheck();
  if(state[STATE_CONF_X] != object.getx() &&
     state[STATE_CONF_Y] != object.gety())
    return TRUE;
  return(Atombase::Interrupt());
}

Point GoTo::GetPoint(){
  return object;
}
```

```
/*************************************************************/
/* Orient.h                                                  */
/*-----------------------------------------------------------*/
/*     Class that acts as an interface for Orient functions  */
/*                                                           */
/*                                                           */
/*   Created by:Stephanie Wojtkowski                         */
/*   Created on:7-27-1999                                    */
/*   Last Modified: 7-27-1999                                */
/*************************************************************/

#ifndef __ORIENT__
#define __ORIENT__

#include "GRobot.h"
#include "PlanFiles.h"
#include "CMap.h"

class Orient : public Atombase{
 private:
  Point object;      /*The position of the object to follow*/
  float angle;       /*angle between point and the robot*/
 public:
  Orient(Point p, CRobot * r, float speed);
  int Interrupt();
};

#endif
```

```
/*************************************************************/
/* Orient.cpp                                                */
/*-----------------------------------------------------------*/
/*       Member functions for class Orient                   */
/*                                                           */
/*                                                           */
/*   Created by:Stephanie Wojtkowski                         */
/*   Created on:7-27-1999                                    */
/*   Last Modified: 7-27-1999                                */
/*************************************************************/

#include "Orient.h"
#include <math.h>

Orient::Orient(Point p, CRobot * r, float speed){
  float x, y, Ptheta;

  object = p;
  Atombase::Atombase(r);
  x = object.getx() - state[STATE_CONF_X];
  y =  object.gety() - state[STATE_CONF_Y];
  Ptheta = atan(y/x);
  if(y >= 0 && x < 0)
    Ptheta += PI;
  else if(y < 0 && x < 0)
    Ptheta += PI;
  else if(y < 0 && x >= 0)
    Ptheta += 2*PI;
  if (Ptheta > 180)
    Ptheta -= 360;
  angle = Ptheta - State[STATE_CONF_STEER];
 if (angle > 180)
    angle -= 360;
 if(angle>=0)
  Control c(speed, -speed);
 else
   Control c(-speed, speed);
 Atombase::SetUt(c, 0);
 Atombase::SetTimeVal(1000);
}

int Orient::Interrupt(){
  if(State[STATE_CONF_STEER]==angle)
    return TRUE;
  Atombase::Interrupt();
}
```

```
/*****************************************************************/
/* GoTo.h                                                        */
/*---------------------------------------------------------------*/
/*    class that acts as an interface for GoTo functions         */
/*                                                               */
/*                                                               */
/*   Created by:Stephanie Wojtkowski                             */
/*   Created on:7-27-1999                                        */
/*   Last Modified: 7-27-1999                                    */
/*****************************************************************/

#ifndef __GOTO__
#define __GOTO__

#include "GRobot.h"
#include "PlanFiles.h"
#include "CMap.h"
#include "NClient.h"

class GoTo : public Atombase{
 private:
  Point object;      /*The position of the object to follow*/
 public:
  GoTo(Point p, float s, CRobot * r);
  int Interrupt();
};

#endif
```

```
/**************************************************************/
/* GoTo.cpp                                                   */
/*------------------------------------------------------------*/
/*           member functions for class GoTo                  */
/*                                                            */
/*                                                            */
/*   Created by:Stephanie Wojtkowski                          */
/*   Created on:7-27-1999                                     */
/*   Last Modified: 7-27-1999                                 */
/**************************************************************/

#include "GoTo.h"

GoTo::GoTo(Point p, float s, CRobot * r){
  Atombase::Atombase(r);
  object = p;
  Atombase::SetUt(Control(s,s) c, 1);
  Atombase::SetTimeVal(100000);
}

int GoTo::Interrupt(){
  get_rc();
  if(state[STATE_CONF_X] == object.getx() &&
     state[STATE_CONF_Y] == object.gety())
    return TRUE;
  return(Atombase::Interrupt());
}
```

```
/*************************************************************/
/* Stop.h                                                    */
/*-----------------------------------------------------------*/
/*  class that acts as an interface for the Stop functions   */
/*                                                           */
/*                                                           */
/*   Created by:Stephanie Wojtkowski                         */
/*   Created on:7-27-1999                                    */
/*   Last Modified: 7-27-1999                                */
/*************************************************************/

#ifndef __STOP__
#define __STOP__

#include "GRobot.h"
#include "PlanFiles.h"
#include "CMap.h"

class Stop : public Atombase{
 public:
  Stop(CRobot * r);
};

#endif
```

```
/**************************************************************/
/* Stop.cpp                                                   */
/*------------------------------------------------------------*/
/*               Stop functions                               */
/*                                                            */
/*                                                            */
/*   Created by:Stephanie Wojtkowski                          */
/*   Created on:7-27-1999                                     */
/*   Last Modified: 7-27-1999                                 */
/**************************************************************/

#include "Stop.h"

Stop::Stop(CRobot * r){
  Control c(0, 0);
  Atombase::SetUt(c, 0);
  Atombase::SetTimeVal(1000);
  Atombase::Atombase(r);
}
```