

RAFS - Robot Aided Feng Shui



Project Design Document (DD)

Peter Motykowski-|Bradley White-|J.D. Pohlman-|Matt Allen

Table of Contents

| | |
|--|----------|
| 1. General Description | 2 |
| 1.1 Purpose of the Project..... | 2 |
| 1.2 Scope of the Software | 2 |
| 1.3 Assumptions, Constraints, and Risks..... | 2 |
| 1.4 Definitions, Acronyms, and Abbreviations..... | 2 |
| 1.5 References..... | 2 |
| 1.6 Overview of the Document..... | 2 |
| 2. Project Standards and Conventions..... | 2 |
| 2.1 Design Standards..... | 2 |
| 2.2 Documentation Standards | 2 |
| 2.3 Naming Conventions | 2 |
| 2.4 Programming Standards | 2 |
| 3. Software Development and Visualization Tools | 2 |
| 3.1 tacti.cs.siu.edu | 2 |
| 3.2 Saphira 8.1 | 2 |
| 3.3 GCC / Make / PuTTY | 2 |
| 3.4 CodeWarrior | 2 |
| 3.5 C*Forge | 2 |
| 3.6 Microsoft Visual Studio C++ 6.0..... | 2 |
| 3.7 VNC – Virtual Network Computing / Cygwin..... | 2 |
| 3.8 Microsoft Visio 2002 | 2 |
| 3.9 AutoCAD 2002 | 2 |
| 3.10 Rational Rose | 2 |
| 3.11 Microsoft Office XP | 2 |
| 4. Design Details..... | 2 |
| 4.1 Feature Relationships..... | 2 |
| 4.2 Diagrams..... | 2 |
| 4.3 Class and Object Design | 2 |
| 4.4 Networking and Database Information..... | 2 |

| | |
|------------------------------------|----------|
| 4.5 Lifecycle Model | 2 |
| 4.6 Test Plan | 2 |
| 4.7 Project Prototype | 2 |
| 5. Attachments..... | 2 |

1. General Description

1.1 Purpose of the Project

The RAFS project will consist of a hardware/software solution to a problem set of physical objects. In our case, these objects are rolling chairs that are often scattered around the SIUE senior project laboratory (EB2029). Our solution will be comprised of one or more robots controlled by custom software developed by the RAFS team members. Ideally, the robot will be able to identify unorganized chair placement and move the chairs to open computer desks. While doing this, the robot may be required to deal with unexpected objects or situations. These exceptional conditions will be discussed later in this document.

1.2 Scope of the Software

Project Scope

The RAFS project is primarily concerned with the core functions of object recognition and placement. Although we aim to handle as many exceptional conditions as possible, only a limited subset of error states will earn our attention.

In Scope

Within this section we will define goals, deliverables and progress milestones. We plan to structure our work in such a way that each milestone will yield a fully functional, and to some degree, tested, software module.

Goals

- Recognizing an object and being able to determine whether or not it is a chair or desk.
- Maneuvering in an area and being able to avoid objects.
- Identifying an empty desk.
- Enabling the robot to grasp and move chairs.

Deliverables

Each goal will produce a completely functional standalone software module. Using this approach will ensure quality-tested platforms to begin each next phase of development. Using this development approach will help expand the SIUE repository of example robotics source code.

Exceptional Conditions

Inappropriate behavior can result when the robot encounters objects and/or situations with which it is not programmed to deal. We plan to tend to certain situations, which may interfere within an ordinary operation cycle. The objects/situations causing this behavior can be sorted into two categories, dynamic and static objects/situations.

Dynamic Objects/Situations

Objects that fall into this category are those that do not belong to the original layout of EB2029. The room is equipped with collapsing wall dividers and rolling tables. At any time these objects may be placed in various locations throughout the room. As a result, we may add facilities to determine room layout during each operation cycle. Several objects may not necessarily move, however, they may produce a situation the robot needs to deal with. Loose cabling or carelessly placed objects, i.e., temporarily stored boxes, could become obstacles for the robot. Facilities must be in place to avoid such objects and/or correct situations caused by them, i.e., wheel caught in wires. In worst case scenarios it would be appropriate to power down the robot and terminate the operation cycle.

Static Objects/Situations

Objects that fall into this category are those that belong to the original layout of EB2029. The room is equipped with several computer stations that we will refer to as desks. Supporting the ceiling are poles that reach from floor to

ceiling. Two large tables reside in the center of the room. Although these tables may be moved, we will assume that they have been returned to their proper places. We make this assumption based on the markings on the floor that indicate the appropriate placement of the tables. It is fairly clear that the tables belong in this position and should not be moved for any extended period of time. At certain parts of the floor are small mound-like wire covers. The robot alone may roll over the covers with ease, but there may be a concern while the robot is moving a chair. We will determine at a later time if the covers are something that must be avoided. If confronted with an unrecognizable object, the robot will either avoid the object or enter the idle state until assisted by the user.

Random Objects/Situations

In some instances the robot may encounter foreign matter while performing an ordinary operation cycle. In the event that these objects/situations are unpredictable, we must not be responsible for the behavior of the robot. An example of this would be water or some other foreign matter either on the floor or in the path of the robot. These conditions are beyond the scope of arbitrary collision detection.

Environment Issues

There exist several variables within EB2029 that may play a role in the ordinary operation cycle. Room lighting may influence the accuracy of sonar and/or laser guided object recognition elements. In addition, room temperature may affect robot operation, although this case is much less likely. We must also consider the possibility of the floor being waxed on occasion. This could pose a problem with the steering and wheel elements.

Out of Scope

To precisely define what we are trying to achieve we must sometimes define what we are not trying to achieve.

Exceptional Conditions

While it would be ideal to accommodate all exceptional conditions, it would be overly optimistic to strive to do so. We will only be able to handle conditions that are of the highest priority. High priority situations are those in which the robot, SIUE equipment, and/or spectators are in immediate danger.

1.3 Assumptions, Constraints, and Risks

Assumptions

- The project will be performed in the Engineering Building (EB2029).
- The door to EB2029 will be closed during the ordinary operational cycle. This assumption is in agreement with the Department of Engineering's policy regarding the secure status of this room.
- There will be a limited amount of people in the room during the ordinary operation cycle. We do not anticipate more than Dr. White, the RAFS group members and a few spectators.
- The square tables in the middle of the room will remain stationary.
- The computer desks will remain stationary.

Constraints

- The project will be performed in the Engineering Building (EB2029).
- We are limited to robots owned by SIUE. This may include one or both (the acquisition of a third robot is currently pending).

Risks

The project may be too complicated to be successfully completed in the time allotted. However, there appears to be ample time (in the neighborhood of 10 months) indicating this risk may not be a significant issue if consistent progress is made. (Risk assessment: HIGH)

The project may not be able to achieve all goals as set out by Dr. White or those stated in the PDD. Although the PDD process provides the team an opportunity to identify which goals are achievable, necessary, or unattainable. (Risk assessment: MEDIUM)

The robot may pose a damage risk to property in the room such as tables, computers, printers, chairs, or walls. However, the robots have an emergency "OFF" switch. In the event that the robots damage SIUE property, they could always be quickly powered down. The robots also have image processing and object detection capabilities that could be used to avoid damage to objects in the room. (Risk assessment: MEDIUM)

The robot may present a physical threat to people who are in the room at the time of robot operation. On the other hand, the robot will most likely be used after hours, to put chairs back when students have left the room. When the robot is run under these conditions, only the team members and faculty may be present. Also, most operators could get out of the danger's path long before any injury could happen to him or her. Finally, the image processing and object recognition capabilities of the robot could be used in this instance as well. (Risk assessment: LOW)

The robots are shared resources in the School of Engineering and this brings up several issues. First, currently any student in the School of Engineering with an electrical engineering account can access the robot. Second, the robot can be controlled via a LAN/WAN and is susceptible to viruses and other problems that plague networked computers. However, since this risk was initially realized, Dr. Weinberg has been gracious in allocating the smaller robot specifically for our group's senior project use. (Risk assessment: LOW)

1.4 Definitions, Acronyms, and Abbreviations

- **Robot** – The use of this word throughout the document will serve the same purpose as the notation robot(s). This distinction is made to avoid awkward subject verb agreement in sentences. We are unclear at this time whether the project will include one or more robots; therefore this redefinition is absolutely necessary.
- **Ordinary operation cycle** – This term shall represent the following cycle of operation.
 1. Robot executes startup procedure, default chair arrangement procedure selected.
 2. Robot identifies misplaced chairs and empty desks.
 3. Robot places each chair, one at a time, into empty desks.
 4. Robot executes shutdown procedure.
- **Desk** - An empty or chair occupied computer station

- **IDE** – Integrated Development Environment
- **GNU** – GNU is Not UNIX, open source development project
- **CVS** – Concurrent Version System
- **GPL** – GNU Public License
- **HTML** – Hyper-Text Markup Language
- **Cygwin** – A UNIX API system implementation for Win32.
<http://cygwin.com>
- **Pioneer/Frontier2** – The smaller of the two robots
- **XWindows** – A Window-like GUI environment for UNIX platforms
- **XFree86** – A free XWindow implementation that will be used in our project
- **XDMCP** – XWindow Display Manager Control Protocol
- **R&D** – Research and Development
- **PHP** – PHP Hypertext Preprocessor
- **UNIX/Linux** – These terms are used interchangeably throughout this document, dealing with the platform we will be working with
- **NIS** – Network Information Systems
- **Win32** – A term use to describe the Microsoft Windows operating system environment
- **SOENT** – School Of Engineering Windows NT network
- **SSH** – Secure Shell login facility
- **RGB** – A set of integer values describing a color
- **PDD** – Project Definition Document
- **PPD** – Project Plan Document
- **DD** – Project Design Document

1.5 References

- The NORRT project team
- <http://robots.activemedia.com>
- Dr. White
- Dr. Weinberg
- Dr. Umbaugh

1.6 Overview of the Document

The intention of this document is to communicate our final design decisions to our customer, Dr. William White. We will be discussing our entire design plan along with other relevant issues. The software tools being used to develop this project will be discussed in detail along with

refinements to our schedule and test plan. We will address module-to-module communication and classes and objects as well.

We will also provide a divided relationship area that is broken up into three parts: a committed section, a casual section, and a fantasy section. The committed section deals with what requirements our project has to meet to complete the project. Missing any of these requirements negatively affect the course grade. The casual section is what our team hopes to get done. Realistically speaking, we will not accomplish all of the points in the casual section. We will attempt all of them, but our realistic goal is to get at least half of them done. The third category is our "wish list" for this project. We will not even attempt to do these points. They deal with what we wish we could do with the robot for this project. These are nice ideas, but are too complicated to finish in the time allotted for our project.

2. Project Standards and Conventions

2.1 Design Standards

Code will be broken up into different modules. Each module will consist of a header with the programmer of the module, the date, parameters, version number, and a brief description of what the module does. Modules will be broken up into four main categories: movement, chair handling, object recognition, and error logging. The module graph is in the attachments section at the end of this document.

The modules will be designed as follows:

The Saphira API is the very top level of our project. It will be the main interaction for the user, or where the user can call functions (or modules). The only module that the user will be aware of is the Arrange Room module. This module will take care of organizing what functions to call when. It is the main function in our program. It will call Object Recognition, Chair Handling, and Movement.

Object Recognition will have different sub-modules for the laser, sonar, and camera. Each of these sub-modules will call some sort of routine for a chair, desk, and an "other" object. The other is what the robot will do with any other type of object, which is not recognized as a chair or a desk. A module, Error Logging, will monitor each of these routines.

The Chair Handling module will call two sub-modules, one dealing with gripping a chair and the other moving with a chair. The sub-

module that deals with moving a chair will also be used in the Movement module. Error Logging will also monitor these sub-modules as well.

Movement is going to be a major part of this project, so we decided to make it a module. The Movement module has two sub-modules, moving with a chair and moving without a chair. Moving with a chair has two routines - one that makes sure the collision detection is off, and the other that deals with a chair when it is at an empty desk. Moving without a chair has three routines – one that makes sure the collision detection is on, one that finds a chair, and one that finds an empty desk. The Error Logging module again monitors all of the routines.

2.2 Documentation Standards

The group leader will assign sections of the documents for the group to work on. He will base his assignments on areas of interest expressed by each group member. If a group member expresses no interest, some remaining sections will be assigned. When a group member finishes his sections, he will post them on the ftp site, following this naming format:

pdd_Document_1.3.1.doc

Where “pdd” stands for Project Definition Document (or ppd – Project Plan Document or dd – Project Design Document). Documents are started at version 1.0 and the minor section is incremented when sections are completed. If a release is made with grammar and spelling corrections and no additional sections, a release field is added, i.e., 1.3.1.

Documents will be stored on the ftp site, <ftp://csfs2.siu.edu/sp/s02q2>. A directory will be created for each sub-area of the project. For example, a folder is made for each type of document, using the names pdd/, ppd/, and dd/, respectively. As progress is made on each document, older versions of work products will be placed in an archive folder, i.e., /pdd/archives.

Minutes will be taken and kept up-to-date by the Lead Documenter. The Design Document will have the most current information about the project. The older documents will show how the project has progressed, with changes being updated from one older document to a newer one. For example, one thing that has changed is the definition section. Many more definitions have been added to the Design Document as opposed to the Project Definition Document.

Not much of a user manual is required for our project. It will include the procedure for operating the robot with respect to putting the chairs back. It will explain how to turn the robot on, log into the robot, and how to run the appropriate programs to get the task done that the user wants to do.

Several tools will be used to verify and validate our projects. First, Dr. White will give us valuable feedback in regards to our project's progress. The RAFS team in general will be verifying everything is correct as the work progresses. The Lead Documenter will check over the segments while he is combining everything to one document, and if time permits, someone will check over the final document to make final revisions before the document is turned in.

2.3 Naming Conventions

In addition to document file naming standards, we have introduced file naming standards for source code files as well. Thus far we have been using Hungarian naming conventions to organize our source code files. This method uses variations in the capitalization of letters within a filename to provide readability. An example of this is the directory structure and filename from our project prototype. In this example all classes begin with the name of our project, RAFS. In the future, to ensure short names, this may be removed.

```
./rafsGripper
```

```
./rafsGripper/rafsGripperClose
```

```
./rafsGripper/rafsGripperClose/rafsGripperClose.cpp
```

```
./rafsGripper/rafsGripperOpen
```

```
./rafsGripper/rafsGripperOpen/rafsGripperOpen.cpp
```

```
./rafsGripper/rafsGripperRaiseLower
```

```
./rafsGripper/rafsGripperRaiseLower/rafsGripperRaiseLower.cpp
```

```
./rafsSquare
```

```
./rafsSquare/rafsSquareClockWise
```

```
./rafsSquare/rafsSquareClockWise/rafsSquareClockWise.cpp
```

```
./rafsSquare/rafsSquareCounterClockWise
```

2.4 Programming Standards

Coding Standards are important for many reasons. First, if there is some set of standards, all developers will feel as if they are on the same level. The standard for this project was developed using the wide and varied experience of the RAFS team and also by the international development community as a whole. These standards were developed with the following goals in mind:

- Developers can look at any code and surmise what is going on.
- Developers new to the code can get up to speed quickly.
- Developers new to the API need not develop their own coding style.
- Developers new to the API will not make the same mistakes over and over again as previous development teams have.
- Programmers make fewer mistakes in a consistent environment.

This section of the Design Document is our coding standard for writing C++ code to use with the Saphira 8.1 API for the ActivMedia robot. The range of this section is devoted to coding style, not to code function.

Background

The various experiences of our team, both in and out of the classroom and industry, have developed this standard in an attempt to make this latest project progress smoothly. Are these standards necessary or vital to the fruition of the project? Probably not, but they are meant to help and as developers we will graciously accept help when provided to us. When a standard is composed in a logical matter using reasonable assessments it can also be overbearing or technically lacking. Keep in mind that this project is a team effort, therefore it was easier to develop this standard as a team.

Introduction

This section of the document does not address functional organization. While we have tried to be clear, concise, and complete as possible, this standard cannot be imposed on all situations or circumstances.

The standards listed in this section of the document pertain to this project only. However, we welcome, invite, and encourage other

organization to use our coding standard for their software development projects. We do ask though that notification of the members of the RAFS group and the Southern Illinois University Edwardsville School of Engineering of their intentions to use this section of the design document.

Standards Enforcement

The RAFS team members are great believers in the “Honor System”. There is no committee on enforcement of the coding standard, only peer reviews of one another’s code and the help of other teammates in debugging code modules. This coding system is encouraged for the latter of these two reasons.

Names

Since names are the heart and soul of development, names that are clear, unique, and descriptive are of great value to the development team and to future generations of developers who will be upgrading, updating, and studying our code.

Class Names

Names of classes should be names of things that already exist (both in the computer system and the real world). If we are writing a class to describe desks, we should name the class “Desk”. If we are writing a class for error we should name the class “Error”. *Compound words and names containing more than three words will be avoided at all costs.*

Source Code File Naming Conventions

First, as with most computer files, files require a name, dot separator, and suffix.

- All Source code files will end with a “cpp” or “c” suffix
- All Header files will end with an “h” suffix
- All files will use a dot separator between name and suffix “*.h” or “*.cpp”

Class Conventions

As naming conventions go, the name of all source and header files are left to the discretion of the developers as long as they meet the following requirements:

All file names must be contiguous and contain no spaces.

Allowed:

“Class.h”

Not-Allowed:

“My class.h”

All classes must be singular nouns (objects) that begin with a capital letter

Allowed:

“Desk”

Not-Allowed:

“desks”

All methods for inserting data into a class object will be named with the prefix “set” and the next word will be the name of the attribute(s) whose value is being changed. The name of the attribute being changed must begin with a capital letter. No underscores will be permitted in naming class input functions. All class input functions will be of type void and all parameters passed into the function will be passed by value.

Allowed:

```
void setClientName(string FirstName, string Last Name);
```

Not-Allowed:

```
bool Send_Client_Name(string &first_name,  
                      string &last_name);
```

All methods for retrieving data from a class object will be named with the prefix “get” and the next word will be the name of the attribute(s) whose value is being retrieved. The name of the attribute being retrieved must begin with a capital letter. No underscores will be permitted in naming class output functions. All class input functions will be of type void and all parameters passed into the function will be passed by reference. The parameters for these functions will be the buffers for getting things out of classes.

Allowed:

```
void getClientName(string &FirstName,  
                  string &Last Name);
```

Not-Allowed:

```
bool Gimme_Client_Name(string first_name,  
                       string last_name);
```

Class attributes of standard types will be named by capitalized nouns, with the exception of Boolean types which will be named an adjective or verb ending in the prefix “ed”.

Allowed:

```
string Name  
int Number  
float Average  
bool calculated  
bool measured
```

Not-Allowed:

```
string myName  
int myNumber  
float classAverage  
bool Am_I_measured  
bool Am_I_calculated
```

Documentation of classes will be handled using the following template for C++ source file coding.

```
// class: <class name>  
// description: <a description of what this class is and what it is used  
// for>  
// attributes:  
// <attribute 1> - <description>  
// <attribute 2> - <description>  
// <attribute 3> - <description>  
// <attribute 4> - <description>
```

General Coding Standards

Variables

All variables (except of type bool) will be nouns named with capital letters. Boolean variables will be adjectives or verbs named the suffix “ed” except where they make incorrect grammatical sense.

Underscores will be used only to separate text portions of the variable name from numerical portions of the variable name.

Allowed:

```
bool Named_123
bool Placed_999
```

Not-Allowed:

```
bool Founded123
bool isItName123
```

Declarations

One declaration per line is to be used and followed by a short inline comment to document its significance and use. This convention is to be followed even if variables are of the same type.

Allowed:

```
int Sum;           //Sum of all the incoming data.
int Input;        //The current input from the loop.
char FirstName[80]; //Buffer for user's first name.
char LastName[80]; //Buffer for user's last name.
```

Not-Allowed:

```
int Mikes_Sum;           //Sum for incoming data
char Array_1[80], Array_1[80];
float Mikes_floater_variable; //float variable for stuff
```

Functions & Function Prototypes

Functions and Prototypes will limit arguments to one argument per line followed with a short inline documentation. This convention makes code more readable and easier to follow. Functions will be named with a lowercase verb followed by nouns named with capital letters. Passing by value is encouraged, but the writers of this document understand that sometimes passing by reference is necessary. Try to limit these occasions.

Allowed:

```
float returnAverage(float Table[ ], //Data to find the
                                //average
                    int NumberOfItems //The number of items in
                                //the table
                    bool RoundOff); //decides if returned
```

```
//value will be rounded
//to the nearest int.
```

Not-Allowed:

```
float AverageData(float mytable[],
                  int number, //number of items in
                              //table
                  bool Round_Yes_Or_no); //round off
```

Documentation and in-line comments

Any significant operation or declaration will require a brief inline comment. Insignificant operations not necessary of documentation are those such as updating a pointer to the next character in a string or documentation of the pre-increment or post-increment operators. When in doubt if an operation is significant, ask yourself a simple question. “Would a first year Computer Science Undergraduate understand this?” If the answer is yes, then nothing more is necessary. Otherwise, more documentation is needed.

Necessary: int Total; //Sum total of all values in the set

Not-Necessary: p++ //increment p for the next iteration
//of the loop

The following function documentation shall be used as templates for all functions in this project.

C++ Function Documentation

```
// function name: <name goes here>
// parameters-in: <list them one per line with a short description>
// parameters-out: <list them one per line with a short description>
// parameters-in/out: <list them one per line with a short description>
// returns: <what value does the function return or void>
// description of function: <A full and complete description of what the
// function does and how it doe it will go in this field>
// notes: <any other information a developer finds helpful should go in
// here (e.g., “use this function only for floating point data”)>
```

C Function Documentation

```
/* function name: <name goes here>  
/* parameters-in: <list them one per line with a short description>  
/* parameters-out: <list them one per line with a short description>  
/* parameters-in/out: <list them one per line with a short description>  
/* returns: <what value does the function return or void>  
/* description of function: A full and complete description of what the  
/* function does and how it does it will go in this field>  
/* notes: <any other information a developer finds helpful should go in  
/* here (e.g. "use this function only for floating point data")>  
*/
```

Note that this is the standard for documenting functions and is different from how documentation for functions that access classes will be handled. An example of documentation on how functions that access classes are handled is shown in the class documentation section above.

3. Software Development and Visualization Tools

3.1 tacti.cs.siu.edu

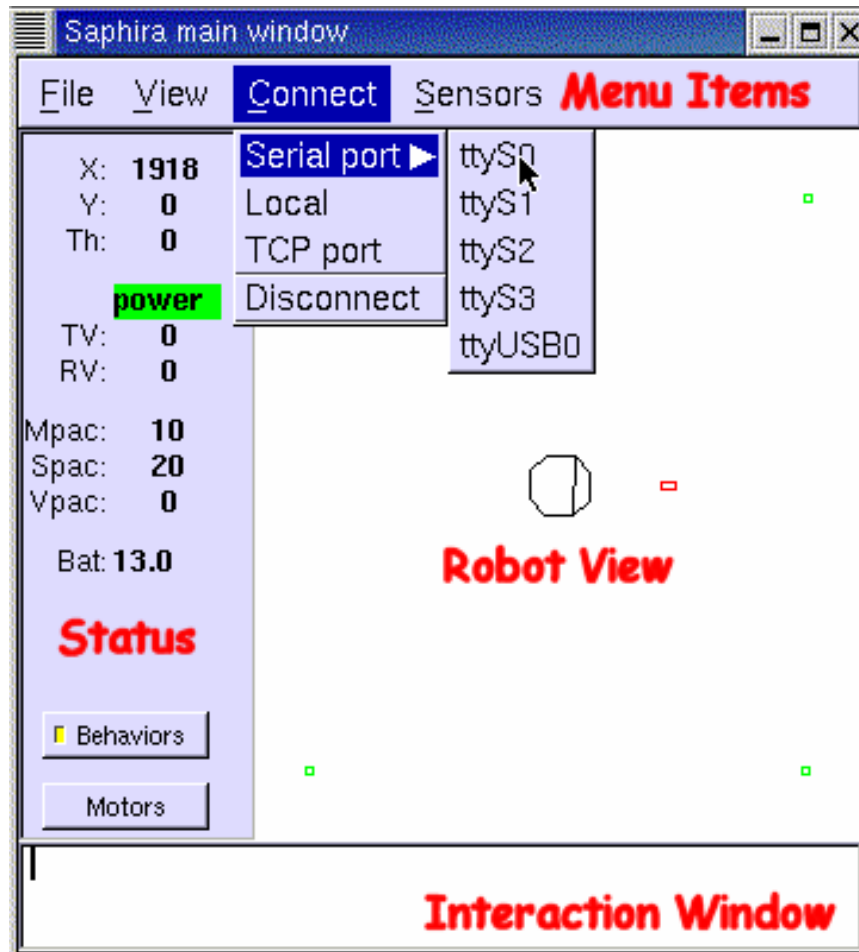
This machine is going to be used to test Linux compatibility of written C++ code. For the sake of convenience, code may be written on a Windows machine using Visual Studio C++ and the Pioneer2 robot simulator. This machine can be accessed via VNC for remote desktop access to the GUI tools packaged with Saphira 8.1. This testbed for code will be an intermediate development point as opposed to authoring code directly on the robot. The decision to do this is based on the recent instability of the robots' onboard computers. In the event that the robot malfunctions, we will not have to worry about potential data loss. In addition tools such as C*Forge and CodeWarrior will be installed for any advanced programming that may take place on this machine.

3.2 Saphira 8.1

This rich sophisticated robotics API will be used for most, if not our entire project. This new release contains a full underlying implementation of ARIA and a richly developed hierarchical class structure. This version of the API was used to develop the feasibility prototype and facilitated the rapid programming approach that was necessary.

Graphical User Interface

Instead of developing our own user interface, we are going to rely on the provided Saphira GUI. This feature-rich environment provides us with all of the needed functionality, along with several extra options that may prove useful. As the Saphira GUI starts, it loads user libraries that can be used within the environment.



Library Files

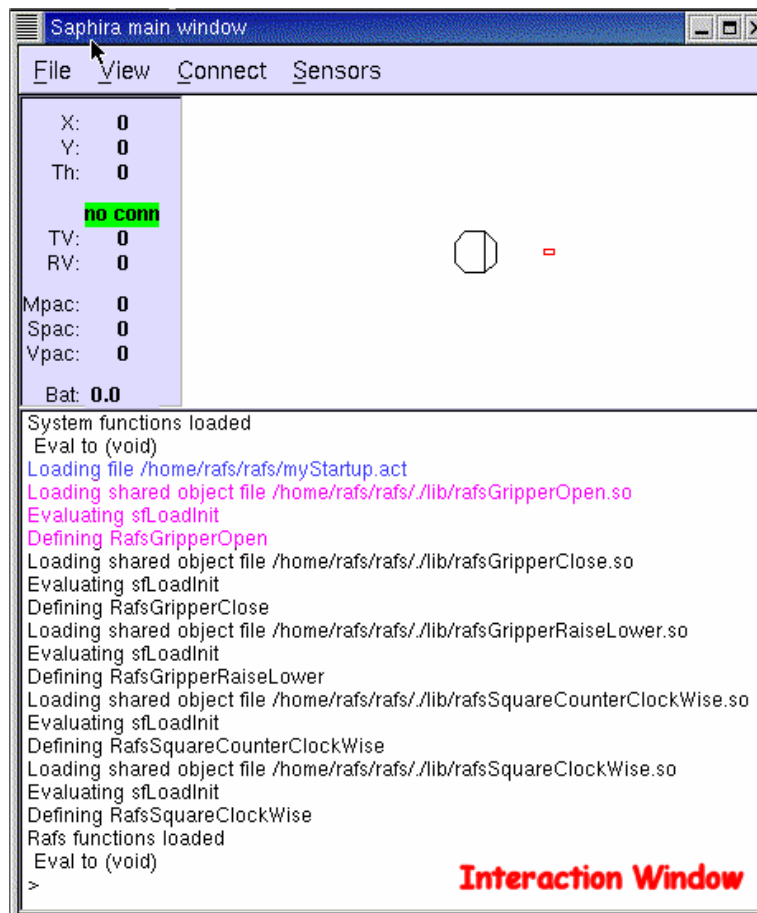
Each functional module of our program will be contained within an ELF 32-bit LSB shared object file. The framework for composing such an object will be reviewed in the detailed description of our prototype code.

Colbert: A Language for Reactive Control In Saphira

This language will be used to some extent when automating the loading of library modules. Since our program will consist of several library

modules, we will need to ensure all are loaded and available upon starting the GUI.

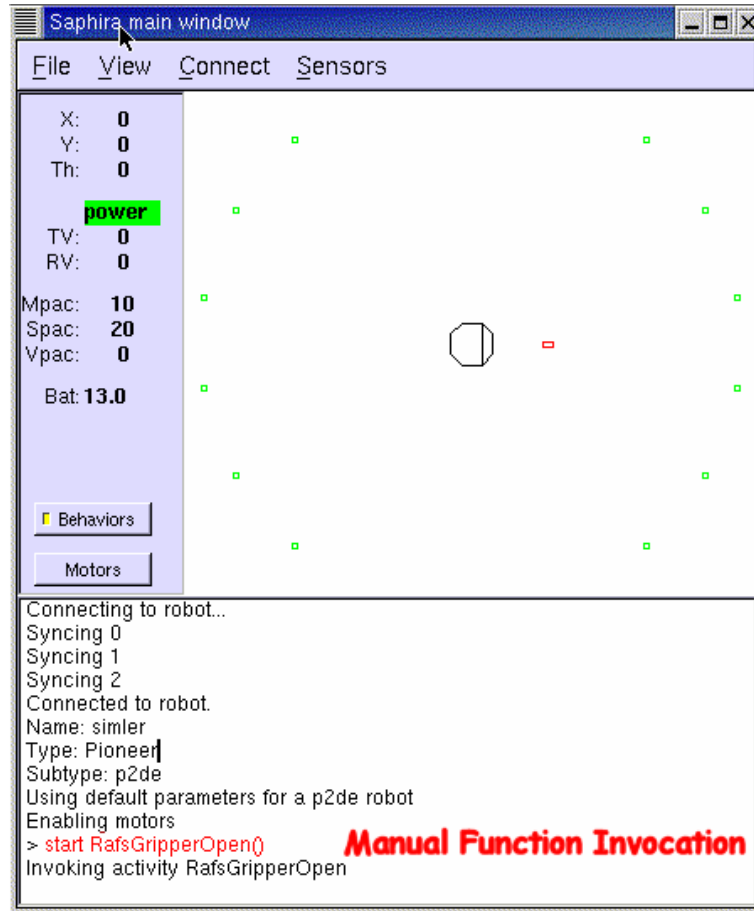
```
// Startup functions loaded after system startup routines
// file: myStartup.act
// These are the libraries used for out prototype
loadlib ./lib/rafsGripperOpen; //See output below
loadlib ./lib/rafsGripperClose;
loadlib ./lib/rafsGripperRaiseLower;
loadlib ./lib/rafsSquareCounterClockWise;
loadlib ./lib/rafsSquareClockWise;
sfMessage("Rafs functions loaded");
```



In addition to automated startup use, Colbert can be used within the interaction window of the Saphira GUI. This window could be used to interact with the program at any point during its operation.

Interaction Window

This window is used as both a means of input and output. Any code using the `sfMessage()` function will be displayed within this window. This window will also be used to manually load or invoke action functions during program execution.



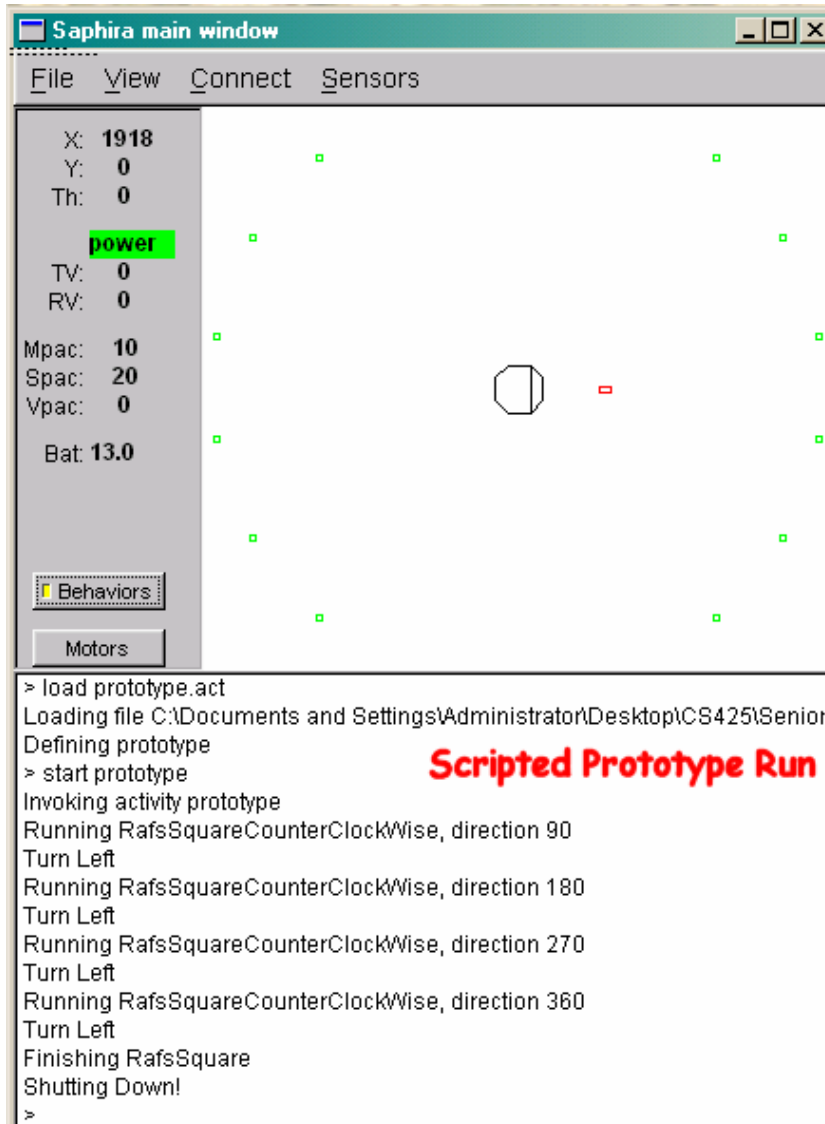
To a certain extent we will use the Colbert language to write batch scripts to execute a series of library functions in a certain sequence. The following script is an example that launches the appropriate functions to complete a counter-clockwise traversal of a 6-foot square with a chair.

```
// This file starts the prototype actions in the appropriate order
// file: prototype.act
act prototype()
{
    start RafsGripperOpen();
    start RafsGripperRaiseLower(0,0);           //Lower gripper to lowest
                                                //point
    start RafsGripperRaiseLower(1,135);        //Raise gripper to
```

```

//appropriate level
//manually set chair in gripper now
start RafsGripperClose();
start RafsGripperRaiseLower(1,35); //Slightly lift gripped
//wheel upward
start RafsSquareCounterClockWise(1829);
}

```

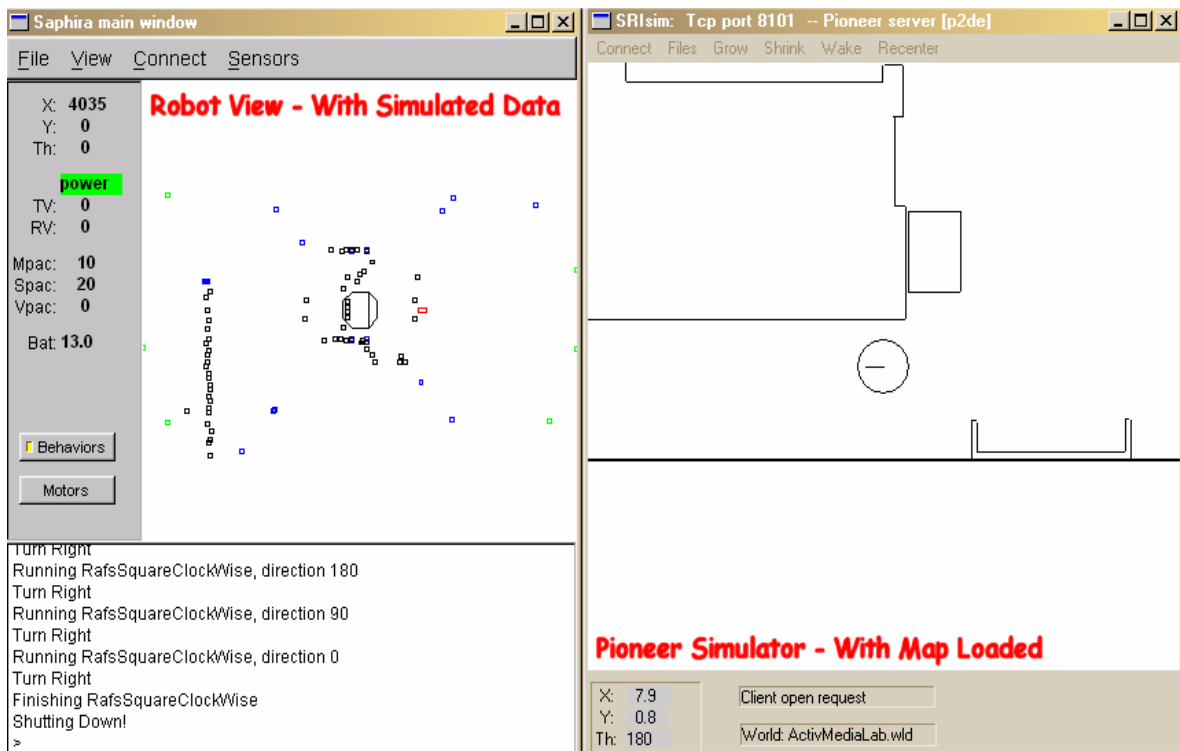


This output lacks messages that would have been returned from the gripper subsystem. This is due to the fact that it was run on the Pioneer robot simulator. This simulator does not have support for external sensory devices by default; therefore, it ignores functions that attempt to access them. As we progress with our use of the simulator we will be able to utilize gripper and laser emulators that return data similar to that expected from physical devices.

The Saphira 8.1 API and GUI are available for both Linux and Win32 operation system platforms. This being the case, we can develop a significant amount of code in the Win32 environment, therefore being able to utilize the Visual Studio IDE. As code is being developed, it could be periodically tested on a Linux Saphira configuration to ensure code can be compiled with the GNU C++ compiler. Although it would be ideal to develop code in the Linux environment at all times, we may use the Win32 option for advanced debugging and testing.

Pioneer Robot Simulator

This simulator is used to imitate a connection to an ActivMedia Pioneer robot. The simulator will be used in both the Linux and Win32 environment when development is not taking place on the actual robot. This will be useful for development tasks that do not require the robot to complete. Also, it will be handy when many team members would like to work on development at the same time or when using off-campus computers.



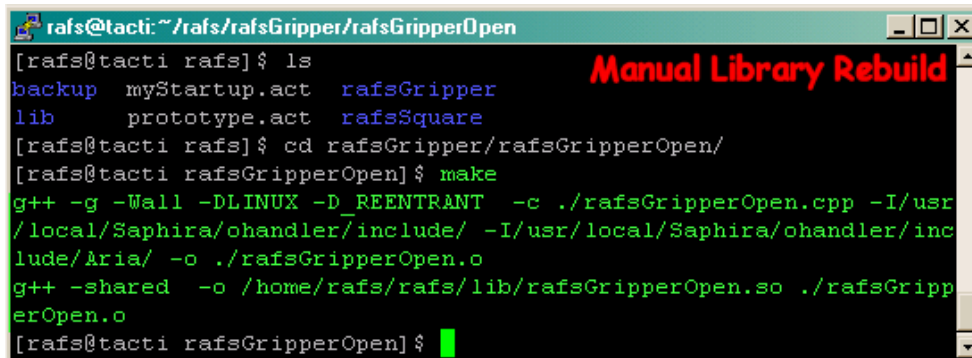
3.3 GCC / Make / PuTTY

To an extent we will be developing code in the console Linux environment when need be. This means of development is considered rather unattractive due to the lack of visual tools to facilitate rapid development. This method will only be used when making small changes to source code or to quickly rebuild a library without having to launch an integrated development environment. We intend on using this method as little as possible to eliminate the chances of human error. Leaving workspace and file management up to the integrated development environment would ensure the integrity of the files in our code base.

GCC – This is the C++ compiler that will be used when building code in the Linux environment.

Make – This is the automated compilation tool that will be used when building code in the Linux environment

PuTTY – This is the Secure Shell remote access tool that will be used to access any Linux based systems used during this project.

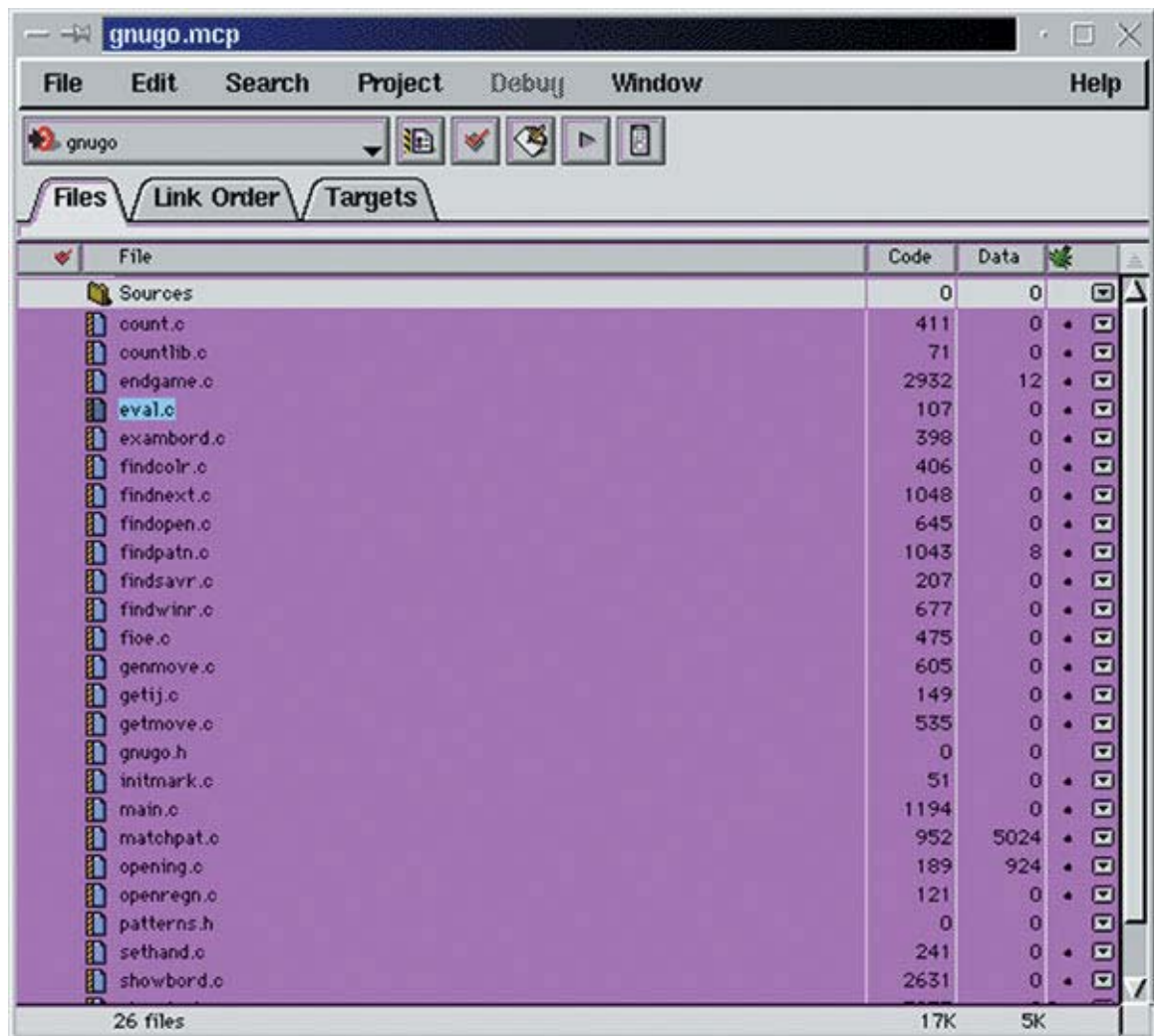


```
rafs@tacti:~/rafs/rafsGripper/rafsGripperOpen
[rafs@tacti rafs]$ ls
backup  myStartup.act  rafsGripper
lib     prototype.act  rafsSquare
[rafs@tacti rafs]$ cd rafsGripper/rafsGripperOpen/
[rafs@tacti rafsGripperOpen]$ make
g++ -g -Wall -DLINUX -D_REENTRANT -c ./rafsGripperOpen.cpp -I/usr
/local/Saphira/ohandler/include/ -I/usr/local/Saphira/ohandler/inc
lude/Aria/ -o ./rafsGripperOpen.o
g++ -shared -o /home/rafs/rafs/lib/rafsGripperOpen.so ./rafsGripp
erOpen.o
[rafs@tacti rafsGripperOpen]$
```

Manual Library Rebuild

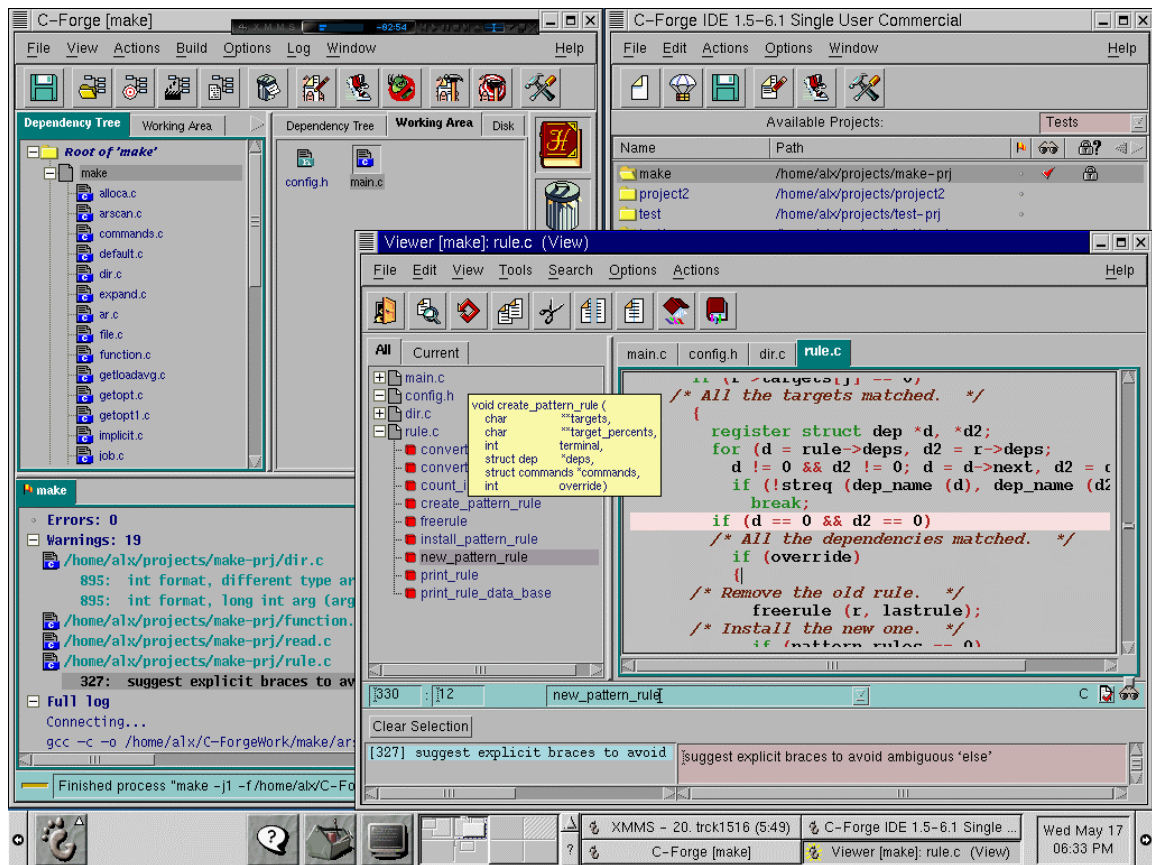
3.4 CodeWarrior

We cannot underestimate the value of a visual development environment. While it would be staying true to the UNIX purist style of development, text editors and command line compilers will not suit our development time schedule. We have referenced an article published by Linux Magazine to help us narrow down our choices of IDEs (http://www.linux-mag.com/2001-04/IDE_01.html). From what the article says, CodeWarrior will be a top choice for our project. It has been around for several years and is part of the curriculum at many universities. We have been unable to obtain an evaluation copy for preview purposes. However, the screenshots and reviews provide adequate information regarding the software suite.



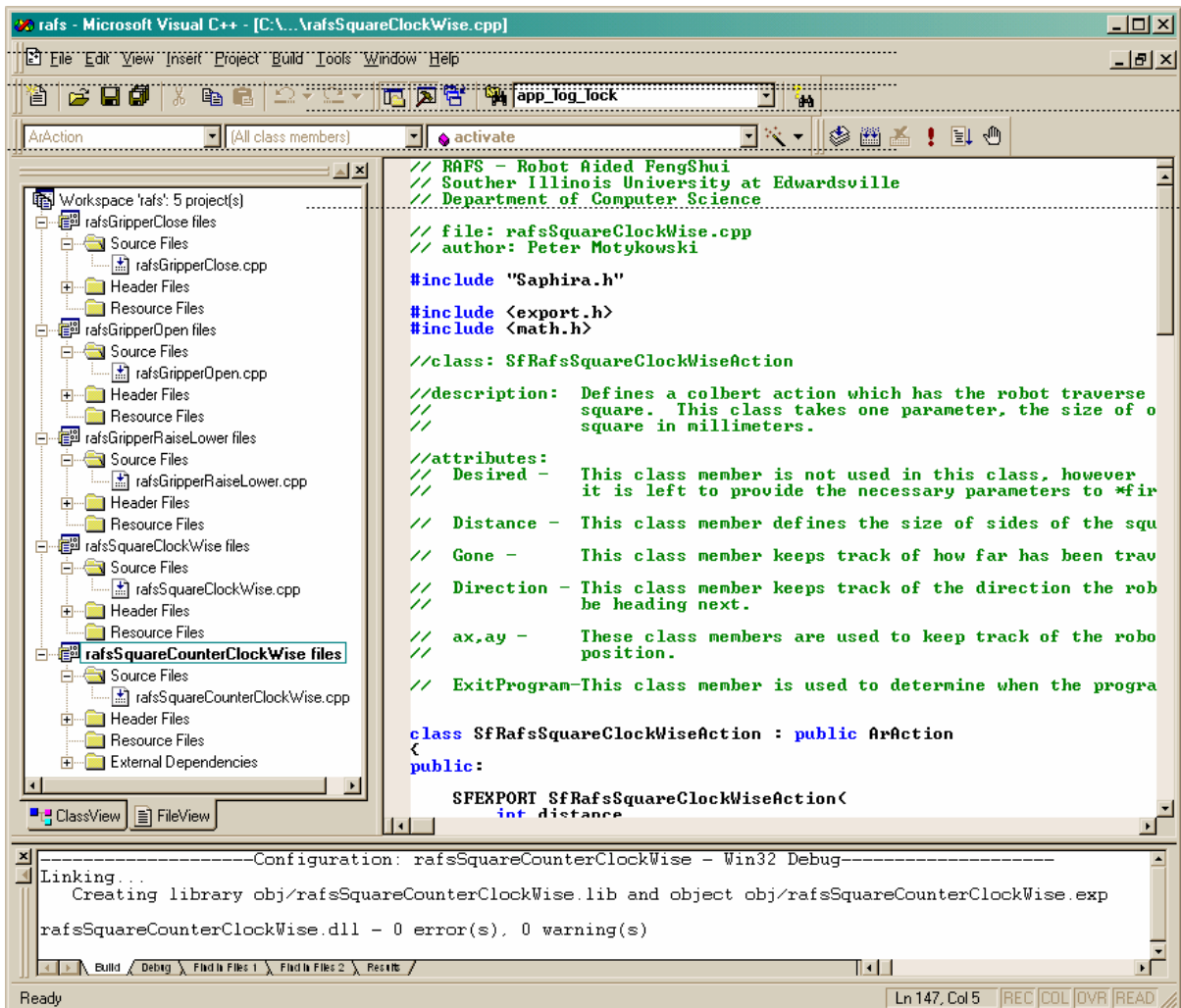
3.5 C*Forge

Another top choice is C*Forge by CodeForge. This IDE seems to be more of a front end to the common set of Linux development tools. This suite was given rave reviews and is said to be an extremely stable development environment. It possesses features such as a visual debugger, *MakeFile* generator and automated CVS support. At this time in our project it is safe to say we would like to use this environment. Although it carries a higher price tag than CodeWarrior, the additional features and stability would be invaluable to us.



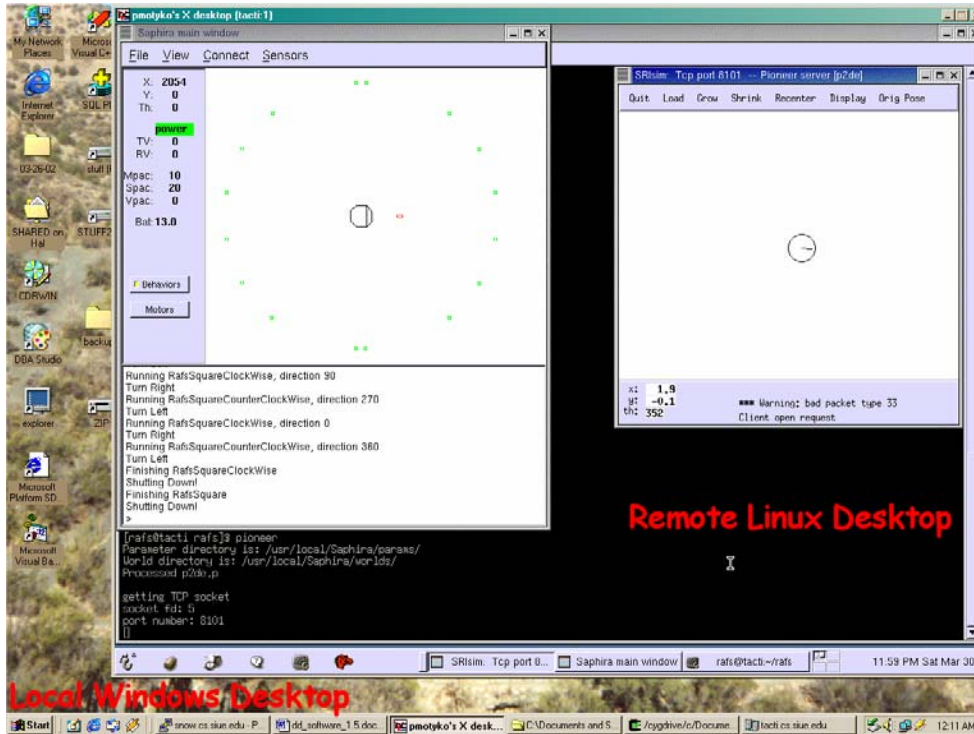
3.6 Microsoft Visual Studio C++ 6.0

We will be utilizing this environment when working with the Saphira 8.1 GUI and simulator on the Win32 platform. As much development as possible will take place using this integrated development environment due to the wealth of debugging tools included. Also, this will aid team members who lack skills needed to be effective developers in the Linux environment. The Visual Studio compiler will produce Win32 dynamic link libraries that can only be used in the Win32 environment. All source code files will need to be rebuilt in the Linux environment to be usable on the robot's onboard personal computer.



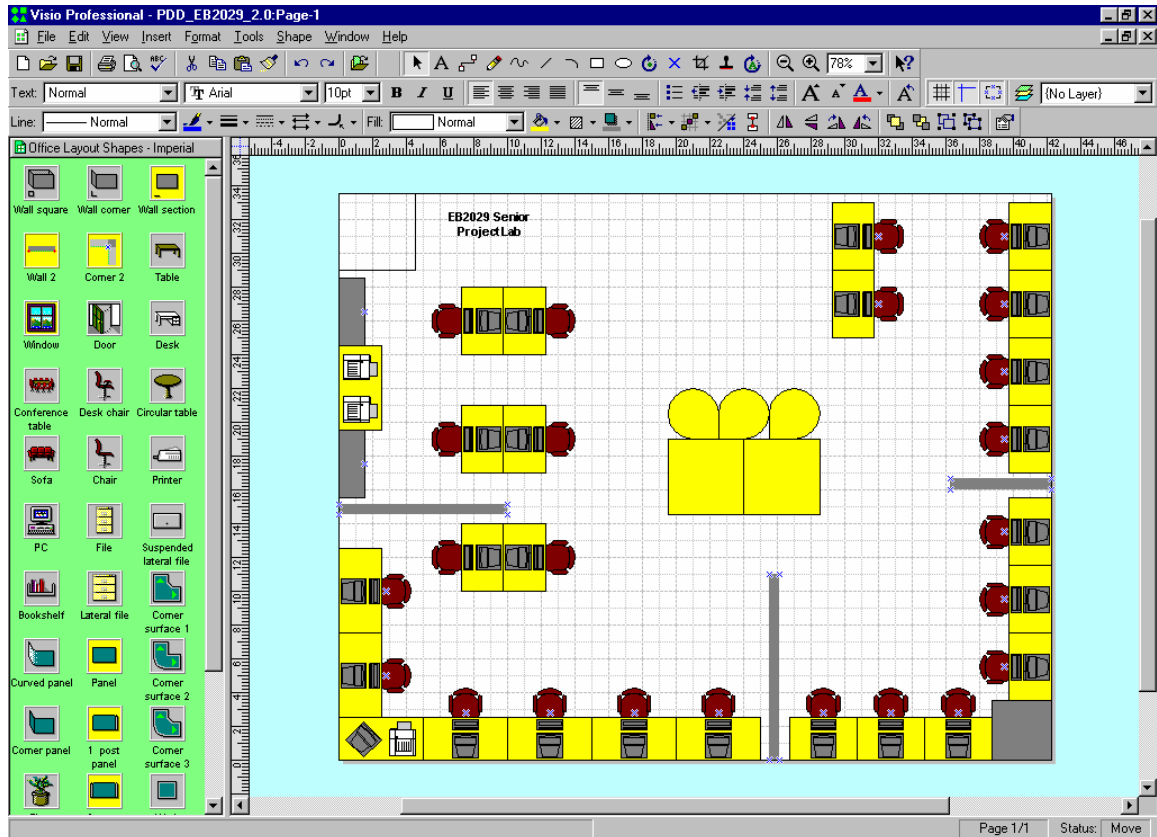
The several different development environments being used are going to introduce issues with version control. We will therefore use a version control system to automate the administration of source code files. Candidate software packages are CVS, RCS or other freely distributed source code control systems.

3.7 VNC – Virtual Network Computing / Cygwin



This tool will be used to access remote Linux desktop sessions from a Win32 client workstation. Again this tool will aid team members in utilizing the Linux computers involved in this project.

3.8 Microsoft Visio 2002



One of the visibility tools the RAFS team will use for the project is Microsoft Visio. Visio is an application that is now owned and maintained by Microsoft. It is part of the Microsoft Office Suite and is used for diagramming and making other visual presentation material.

There are several features and attributes of Visio that can be very helpful to our group in completing our senior project.

First, Visio can be used to make simple block diagrams. This will make it easy to quickly come up with charts for our research and planning.

Second, Visio can be used to design and layout floor plans. This feature will be helpful in quickly documenting the room in which we need to place chairs at their proper station. This feature will also come in handy assuming future growth of our software and allowing the software to be run in rooms other than the senior project room (EB2029).

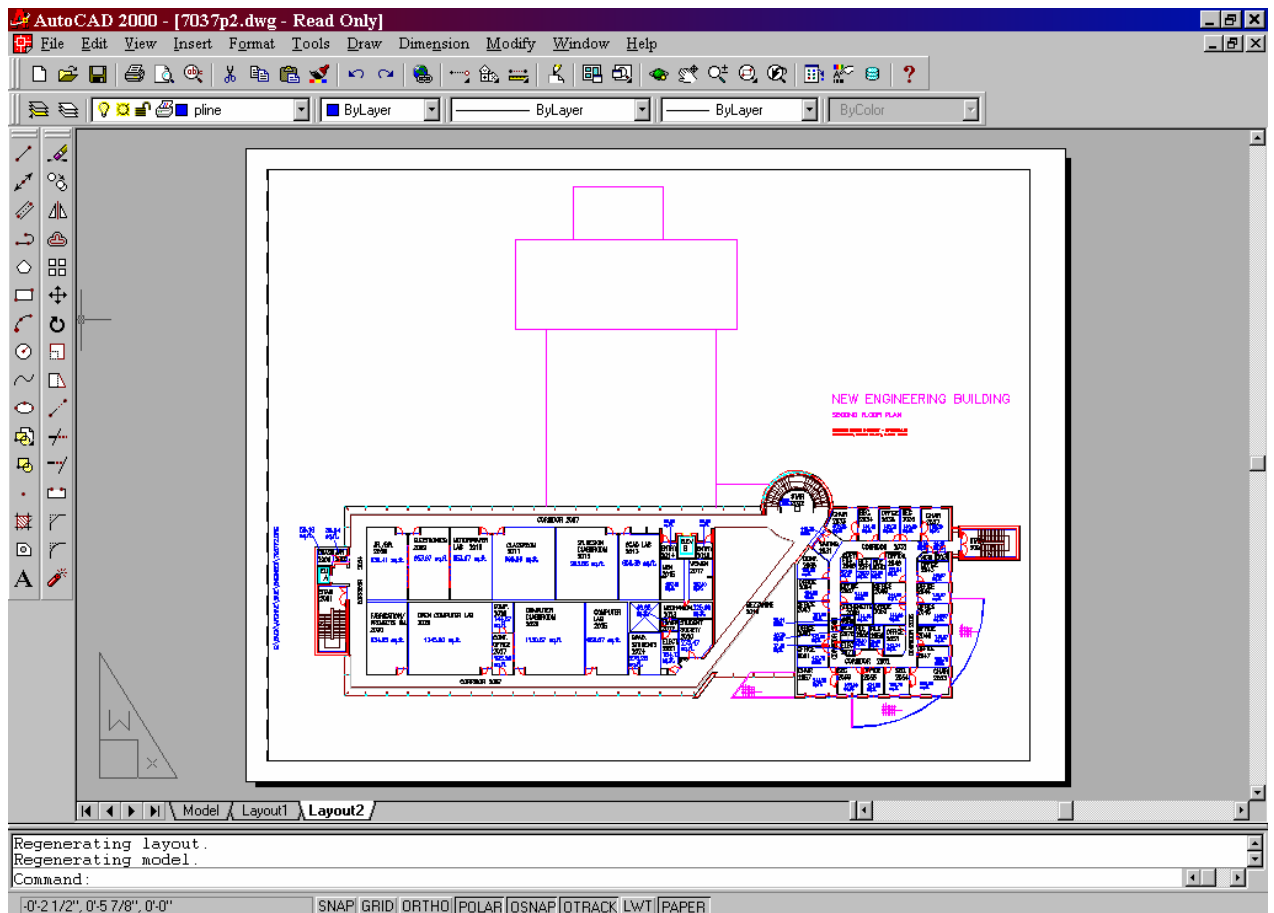
Third, Visio has several business charting features that will allow us to quickly design and print flow charts and document processes (including pseudo-code and algorithms).

Fourth, Visio is designed to work with standard UML notation to quickly and efficiently build Sequence Diagrams, Class Relationships, and State Charts.

Fifth is Visio's short learning curve. It uses a simple drag and drop interface unlike more complex software packages that use complicated "wizards" (such as Microsoft's Project) or dozens-to-hundreds of command line instructions (such as Autodesk's AutoCAD). It also starts up and is ready to operate quickly after launching the program (unlike Rational Rose).

After considering all of Visio's features, capabilities, and short learning curve, we have decided that it would be one of the best software applications to use for the Robot Aided Feng Shui project.

3.9 AutoCAD 2002



Another Visibility tool that we would like to use for our project is Autodesk's AutoCAD. While AutoCAD does not have the short learning curve of Visio or some other graphical development packages, one of our group members is very experienced with it (he is considered an expert) and all computers in the School of Engineering have copies of the latest version already installed.

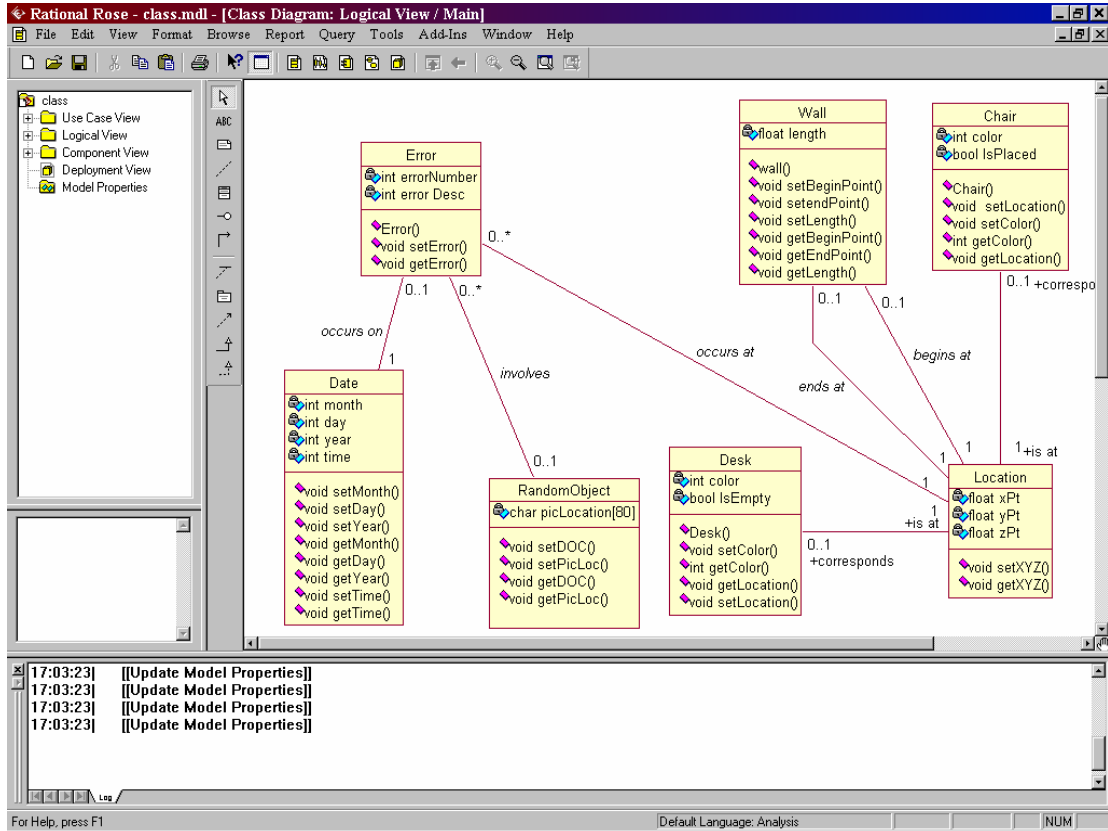
The first of AutoCAD's features is the fact that it uses vector-based graphics. All objects in an AutoCAD file exist as points and lines in space with Cartesian coordinates. This would be very helpful for accurately and precisely documenting a room in which we may have to know the location of every nook, cranny, corner, or major object is. As an added incentive, the Engineering Building was designed using AutoCAD, so any of the rooms that the group may need to document are probably already done.

A second feature of AutoCAD that would be helpful to our group is the fact that it is easy to export to other formats - such as jpg, bmp, and wma. These formats are more commonly used on web pages, documents, and Microsoft Windows applications. Having this tool would allow us to freely export to a more common format, since AutoCAD drawings can be viewed on other computers (on campus and off) without any hassle or worry that some unscrupulous person may try to edit or change the files.

A third feature, the object snap feature, will be beneficial. This feature will come in handy when we need to locate a point that is not easily described or known. Say, for example, on a coordinate plane the midpoint of an arc that is 35 degrees wide. However, with the object snap feature of AutoCAD we can "snap to" that point to find out where it is exactly in the room and program our robot accordingly.

For these reasons we think that AutoCAD would be an excellent tool for documenting and reading documents that relate to the floor plans of the Engineering Building.

3.10 Rational Rose



The Rational Rose software program is a frequently used application throughout the software development industry. Rational Rose is a “model driven development tool” that uses UML (Unified Modeling Language) to make entity relationship diagrams, class diagrams, and also sequence diagrams.

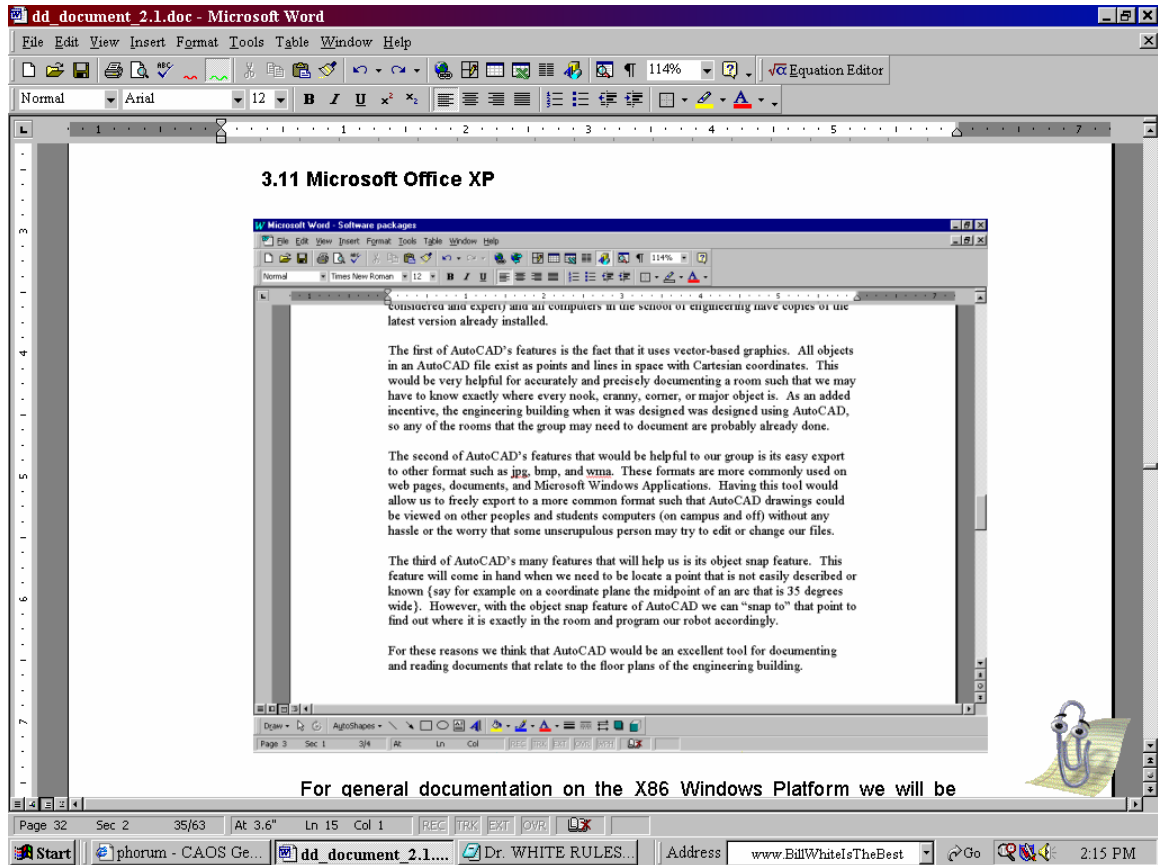
While this package may not be used as much as some of the other software packages used in this document, three quarters of the team have some experience with it and many of the instructors on campus have experience with Rational Rose as well.

Rational Rose has several points that could be very helpful to our project. The first is that it can be used to generate sequence diagrams. We currently are using Visio for this task. The only problem with Visio is that it is not installed on any university computers. It is only installed on one team member’s home computer. We are planning to purchase a legitimate copy to install on university computers.

Rose would also bring its ability to quickly make entity relationship diagrams to our project. We are also currently using Visio but as we had mentioned in the previous point, it is only available to us on a limited

basis. Therefore, we will be using Rose in addition to Visio for doing our entity relationship diagrams.

3.11 Microsoft Office XP



For general documentation on the x86 Windows Platform we will be using (and have been using up to every point thus far) Microsoft Word and the rest of the Microsoft Office XP applications. The XP suite has allowed us to integrate text with jpg, gif, and bmp files to make attractive and functional documents.

4. Design Details

4.1 Feature Relationships

Committed Relationship

Omitting these features in the final implementation signifies that the project is incomplete and/or unsatisfactory. All of these features are fully designed and assigned to specific team members.

Casual Relationship

It's not unrealistic to expect most of these features to be included in the final implementation, but omitting some of them is considered acceptable, as long as the code is implemented in a fashion that lends itself to straightforward expansion to include the omitted features. Prioritized features in this category are fully designed and assigned to specific team members, while non-prioritized features are loosely designed and unassigned.

Fantasy Relationship

None of these features can realistically be expected to appear in the final implementation, although all of them are desirable and serious efforts will be made to ensure that the actual implementation could easily be expanded to include these features. All of these features are loosely designed and unassigned to any particular team member.

Committed Relationships

- We will locate a chair in the room by attaching many color-coded stickers on the chairs. The robot will use its camera to recognize certain colors to be linked to the object as being a chair. The only chairs that will be marked are going to be the ones not already at a desk. We will not have to deal with a chair being at a desk, since that is the goal state of each chair. The robot will just ignore those chairs, which will simply be an object, as opposed to a chair. We can put stickers all over the chair (the front and back of the seat back, the post at the bottom of the chair, each leg of the chair, etc.) to let the robot know that object is a chair. This process will be done before the robot is turned on and ready to operate. An early example of what this looks like follows:



- We will locate a desk in the room in a somewhat different way. We will hard code the desks prior to running the robot. This way, we will know the position of all the desks. This will make it easier to find an empty desk, by going directly to where a desk is, to see if it is empty. To accomplish this, we will position stickers around the front of the desk, along with the back of the desk, to identify an empty desk. These stickers will be a different color than that on the chair. The robot will be able to tell where the chair can be put into the desk because the stickers will stop where the chair cannot move into the desk. For example, the stickers will not be placed above where the computer is under the desk. They will stop before that point. If a chair is at a desk already, no stickers will be placed at the desk, thus making the robot think the desk is empty. The process of placing the stickers on the empty desks will be done before the robot is turned on and ready to operate. An early example of what a desk would look like is this:



- We will take one of these chairs that the robot has recognized and located, and put it into one of the empty desk slots. The robot will find a desk in the room (one with stickers on it) and then search for a chair that is not put up by a desk already. Once it finds a color-coded chair, it will try to find the best path between the chair and the desk. Once the path has been found, the robot will store it in memory. It will then grab the chair and follow the path it has stored. It will end up putting the chair into an empty color-coded desk.
- As far as gripping the chair goes, the robot will assume that the base of the chair will be arranged in a certain way. We will make a leg of the chair face directly north, so the robot will move to those coordinates directly north of the robot to grab the leg. This eliminates the problem of finding where the leg is, since the laser and sonar detect above this point in the chair.
- We will assume that a chair is put up with the desk if it is within three feet of the correct placement. In general, we will push the chair up to the desk, and it is considered to be in the correct place when it stops. This means the chair could be backward in the desk, or not facing it. We will not deal with that in this area. An example of a bad placement, but what would still work follows:



Casual Relationships

- We will locate a chair in the room by placing just one or two stickers on the chair. The robot will use its camera to recognize certain colors to be linked to the object as being a chair. The only chairs that will be marked are going to be the ones not already at a desk. We will not have to deal with a chair being at a desk, since that is the goal state of each chair. The robot will just ignore those chairs, which will simply be an object, as opposed to a chair. We will put one or two stickers on the chair to let the robot know that object is a chair. We can place one piece of paper around the bottom of the post of the chair. We might also have to place a sticker on the end of one leg to get the robot to recognize the end of the leg. This process will be done before the robot is turned on and ready to operate. An early example of what this looks like follows:



- We will locate a desk in the room by a somewhat different matter. We will have the desks hard coded, and will still need a way to recognize an empty desk. In order to do this, we will put one or two stickers on the front of the desk, along with one larger one on the back of the desk. These stickers will be a different color than those on the chair. The robot will be able to tell where the chair can be put into the desk because the stickers will stop where the chair cannot move into the desk. We can put one sticker all the way to the far left of the desk, along with the far right. The process of placing stickers will be done before the robot is turned on and ready to operate. If a chair is at a desk already, no stickers will be placed at the desk, thus making the robot think the desk is just another object in the room, as opposed to a desk. An early example of what an empty desk would look like is this:



- We will take one of these chairs that the robot has recognized and located, and put it into one of the empty desk slots. The robot will find a desk in the room (one with stickers on it) and then search for a chair that is not put up by a desk already. Once it finds a color-coded chair, it will try to find the best path between the chair and the desk. Once the path has been found, the robot will store it in memory. It will then grab the chair and follow the path it has stored. It will end up putting the chair into an empty color-coded desk. It will repeat this procedure for each chair, until either all the desks are full or no more chairs can be found.
- As far as gripping the chair, the robot will be able to find the leg arrangement in any position, as opposed to the legs needing to be arranged in a certain state. It will use image processing to achieve this goal state. We can take a picture of the orientation of the legs, and cut out the rest of the picture (floors, etc). We can then have the robot move to the proper place to grab the chair. This seems like it would require a larger learning curve than some of the other objectives in this category.
- The chair will be facing the desk in the correct position. We will not assume the chair can be facing the opposite direction of the desk. It will have to be facing the correct direction, with respect to the desk. An example of this follows:



- We will log any errors that the robot comes across to a file. This procedure will take care of things like when the robot runs into a wall and stalls, or any other reason it would stall. If, for some reason, the routines would come across any errors (e.g., the collision detection cannot be turned off), the error would be logged to a file.
- We will hope to respond to a collision with a solid dynamic object in the room in some sort or another. If the robot were to run into a table, for instance, we would be required to take some sort of action so the robot is able to correct itself.

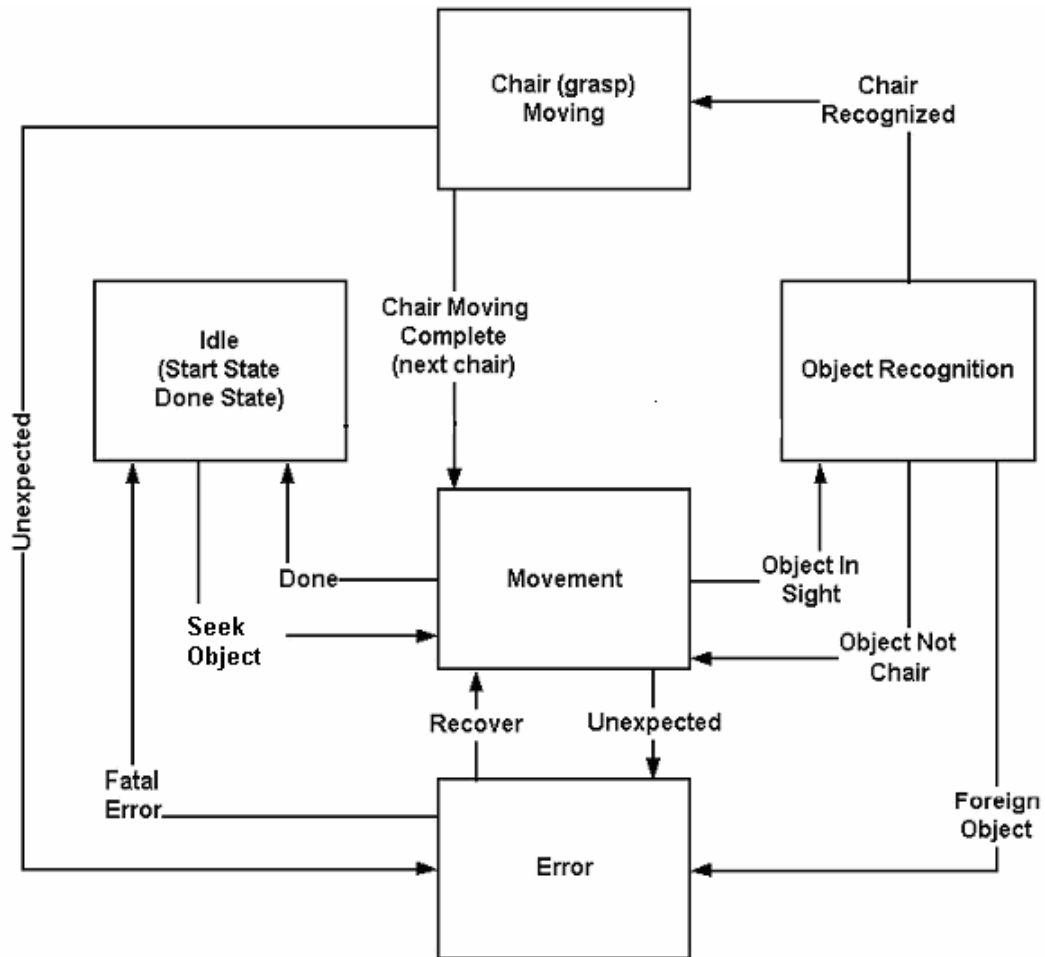
Fantasy Relationships

- Responding to a collision with flexible objects (e.g., wires, cables) in the room.
- The robot can be made to work in any room, as opposed to being restricted in the Engineering Building, room 2029.

- The operator can simply turn on the robot and walk away. This means the job can be left unattended while the robot puts chairs in the correct places. It will deal with ALL problems on its own.
- The robot will deal with movement of objects in the room while it operates. This would allow for people to walk around in the room, and work while the robot is working.
- We will log errors to a database.

4.2 Diagrams

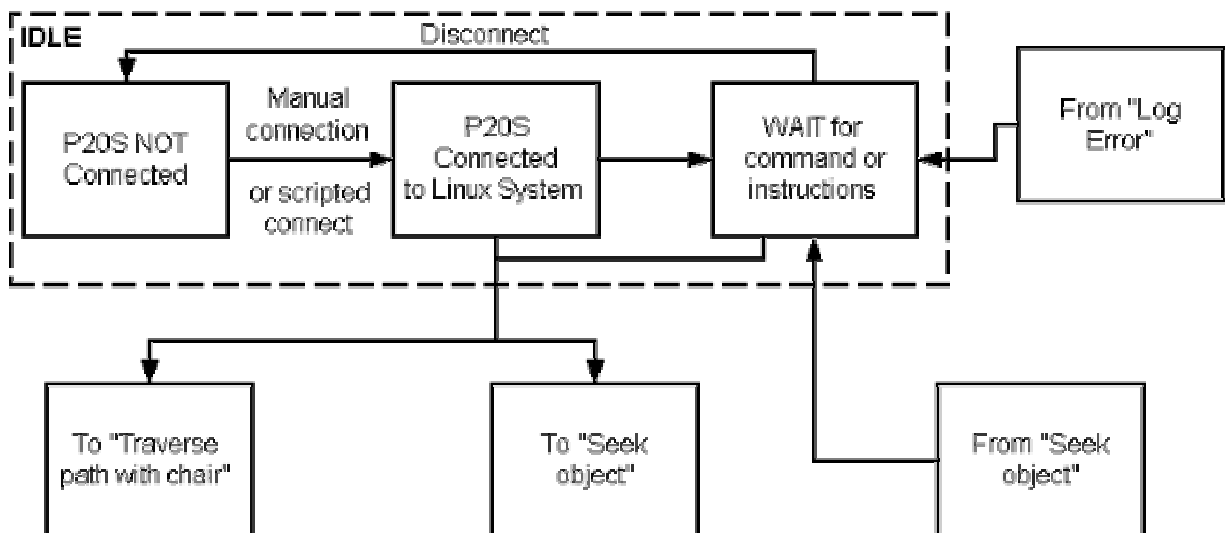
Generalized State Diagram (modified from PPD)



Detailed State Diagram

An elaboration of our state diagram from the Project Definition Document is attached at the end of our document. With some small changes made to the original, we descended into each state, defining the internal characteristics encompassed by each state. As we refined each state, we found a few corrections that needed to be made to the original diagram. This iterative process aided in several design decisions that will be briefed in the following documentation. To help keep our diagram easy to follow we used a color coded numbering system to indicate long connections that would otherwise span complicate the connections between states. For example, the red circle with the number 2 in it coming from the movement state is connected to the red circle with the number 2 of the error state.

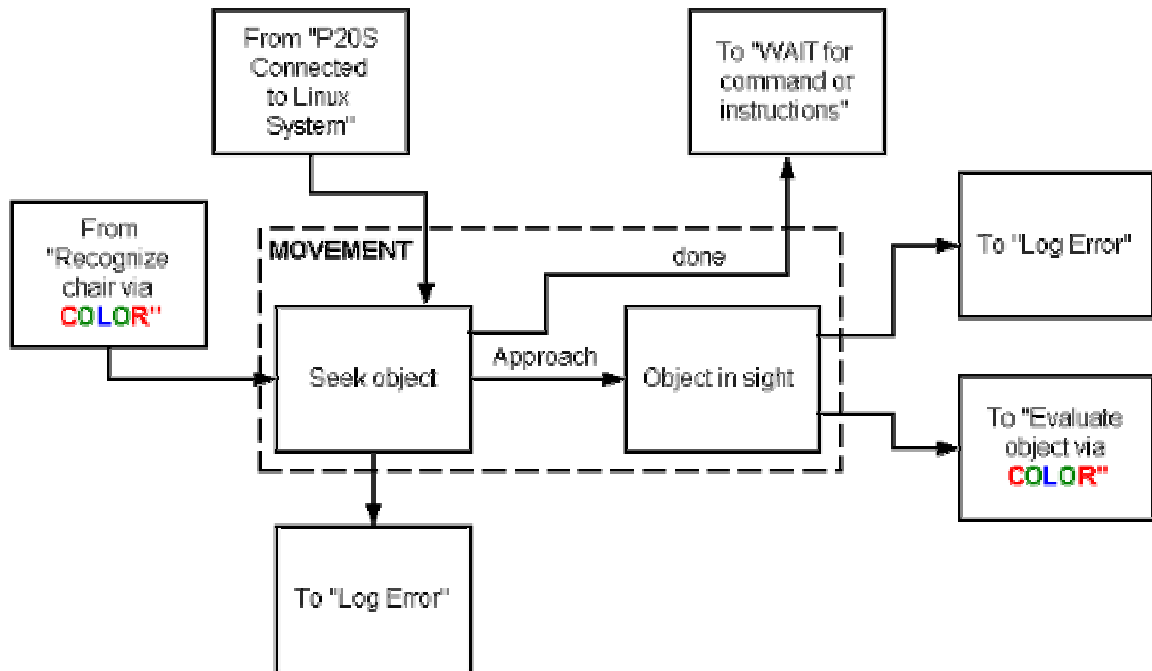
Idle State



- Initially this state will be invoked when the robot is powered up. The robot has power controls for each of the separate computers onboard the robot. When the robot is powered up, the internal robot-controlling computer is powered up and the onboard personal computer running Linux is powered up.
- When both systems are online and operational, the personal computer running Linux can be connected to the internal robot-controlling computer via the serial port (/dev/ttyS0). This connection can be established manually via the Saphira GUI or achieved programmatically by executing a custom binary/script.

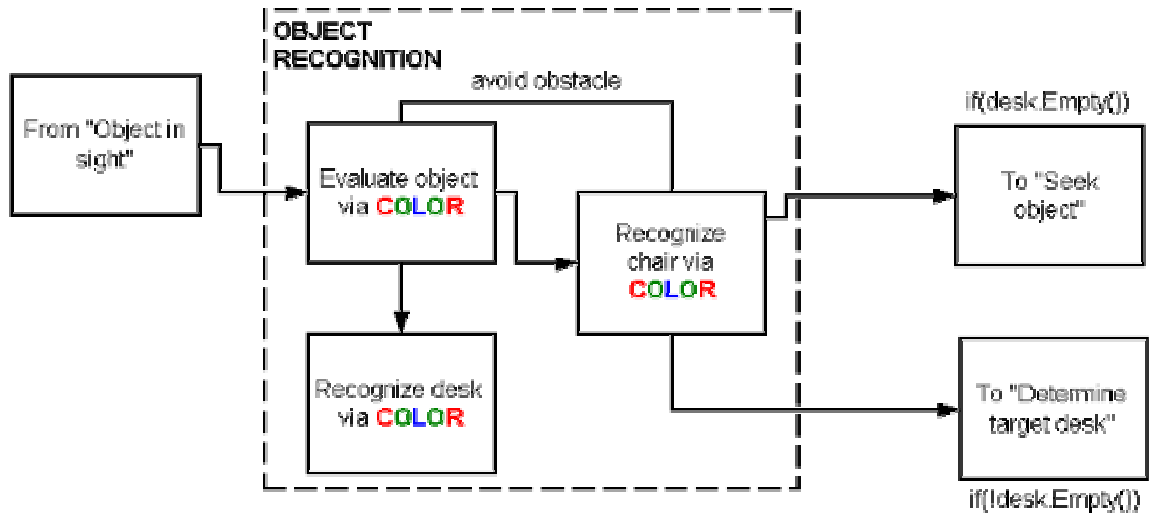
- After achieving a successful connection to the P2OS of the internal robot-controlling computer, the robot is ready to receive commands. At this point a script loading and executing several library modules could be executed, taking the robot out of the idle state into the movement state.

Movement State



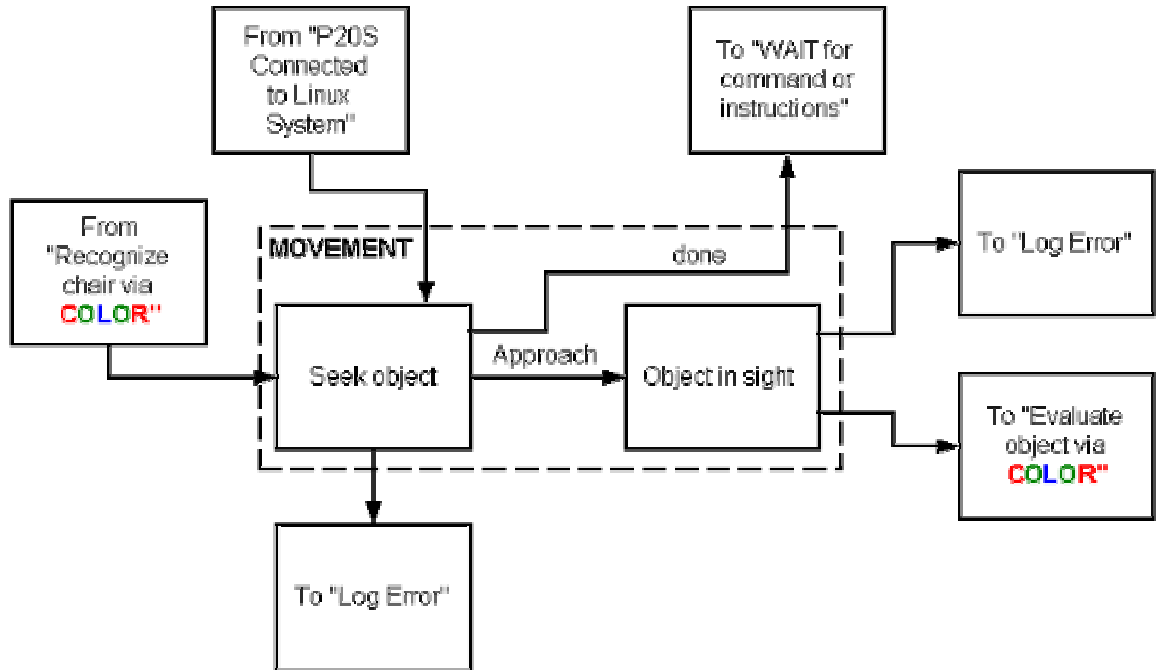
- Upon entering this state the robot will begin seeking an object. An object will become a candidate for object recognition if it seems to have a high concentration of a color that the robot is seeking. The accuracy and triggers of this seeking process will be determined during implementation. If during the seeking the robot becomes stalled, an error state will be invoked attempting to recover from the error.
- When a candidate object is in sight, the robot will approach the object attempting to position itself in such a way to begin object recognition. At this point the movement state will invoke the object recognition state passing to it the data collected on the object in sight.

Object Recognition



- Upon entering the evaluation state, the robot will use the camera to capture several images of the candidate object. A heuristic will be used to analyze the image file searching for recognizable colors. If the predetermined colors are detected to the satisfaction of the heuristic, this object is recognized as a desk or a chair. If evaluation fails the movement state will resume seeking another object.
- If the recognized object is a desk, the data collected from this object will be inserted into the data structure used to maintain the collection of desk objects. Continuing along, the robot will return to the movement state seeking another object.
- If the recognized object is a chair, the data collected from this object will be inserted into the data structure used to maintain the collection of chair objects. If the collection of desks is not empty, the chair movement state will be invoked using the current chair object as a parameter.

Chair Movement



- Upon entering the chair movement state, a route evaluation routine will be performed using the current chair object and the collection of desk objects. This routine will return the optimal path from the current chair object to a specific desk object. If no target desk can be determined, the robot will enter the error state, eventually returning to the movement state.
- When an efficient path is returned, the robot will traverse the path without a chair to ensure the path is accurate and clear of obstacles. Upon successful verification of the path, the robot will return to the current chair object and prepare for chair placement.
- The robot will then grasp the chair and traverse the path that has been verified by the previous step. If need be, the robot will pull the chair to utilize sonar sensors located on the rear side of the robot. These fine details will be discussed later in the document with respect to design decisions. The resulting state will be a successfully placed chair unless some exceptional condition occurs along the way. With this chair placed, control will be returned to the movement state and the robot will continue to seek objects.

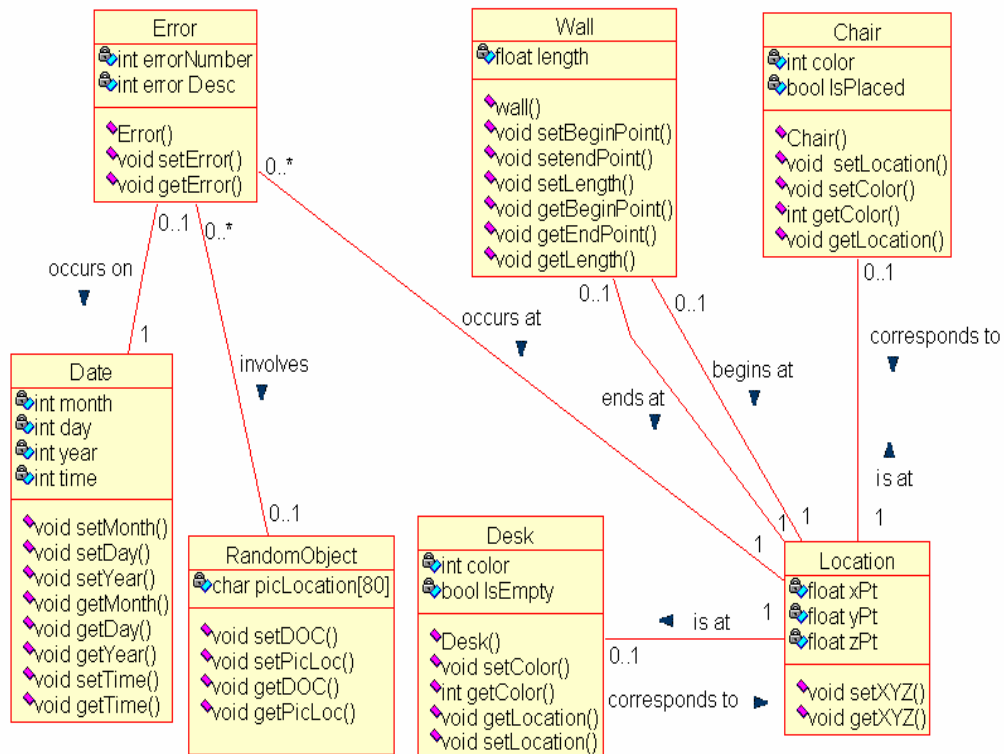
Sequence of Events:

- Turn on robot, loading program
- Robot moves, seeking an empty desk
- Robot recognizes desk, marks coordinates
- Robot moves, seeking a chair
- Robot recognizes chair, marks coordinates
- Robot determines path from chair to desk
- Robot moves chair to desk, following the determined path
- Robot repeats procedure until all chairs are at a desk, or all desks are full

Sequence Diagram

The Sequence Diagram is added in the attachment section at the end of the document.

Class Diagram



Class Relations

Error to Date (0..1 to 1)

A single error can be logged on a single date. However, a date can go by with no error. With time being the smallest attribute in the Date class and the use of only one robot, each Date instance becomes unique.

Error to Location (0..* to 1)

Many errors can occur at a single location. The date of the error makes each occurrence unique. In addition, a certain instance of Location may be associated with many errors.

Error to RandomObject (0..* to 0..1)

Many errors can occur at a single random object. But an error does not have to involve a random object. It may be caused by some other object.

Location to Desk (1 to 0..1)

Only a single desk can occupy a given space obviously, so why does a Desk only occupy a single location? It is true that a Location is only a point, so in all reality it occupies more than one location. For our purposes, we will represent the space a Desk occupies as a single Location. A location does not have to include a desk though.

Location to Chair (1 to 0..1)

A chair will be dealt with much the same way as a desk. Keeping track of a single location of a point will allow us the simplicity of having the robot return to where it found the exact same coordinates it found the chair every time. But, as in the desk scenario, a location does not have to include a chair.

Location to Wall (1 to 0..1)

A wall is at a single location, with a begin point and an end point. We have this represented as two lines on the graph. All locations do not correspond to a wall, so it can be a zero to one relationship.

4.3 Class and Object Design

Class Descriptions

Chair

The Chair class is a representation of the very objects we will be placing in their proper locations at a Desk. The class consists of a constructor, setColor, setLocation, getColor, and getLocation methods as well as the int Color, Location XY, and bool Placed.

Chair Public Methods:

```
public Chair::Chair()
```

The constructor initializes an instance of Chair to Color(-1), XY(-1,-1,-1), IsPlaced(false). XY must first instantiate an object of type Location, using its constructor for initialization of this object.

```
void Chair::setColor(int color)
```

The setColor method defines the color that will be used to represent an instance of Chair. All Chair objects should have their color attribute set to the same value since we will be using a single color to describe objects of this type.

```
void Chair::setLocation(Location XY)
```

The setLocation method sets the Location object of the corresponding instance of Chair. This method will be invoked if the position of the chair changes from that when it was encountered. An assumption we will make is that once a chair is located and a location is stored, the chair will not be moved by any other means than the robot grasping it for movement or placement. This will save us from having to locate chairs several times, saving a significant amount of time. It is also an important note that we do not uniquely identify chairs by any other attributes; however, location is one way to give a degree of uniqueness since no two chairs can occupy the same Location.

```
int Chair::getColor()
```

The `getColor` method returns the `Color` value of the corresponding instance of `Chair`. This attribute will be used to visually identify a `Chair` object.

```
void Chair::getLocation(Location &loc)
```

The `getLocation` method will pass a `Chair Location` back by reference. Our committed deliverable is to locate and place a single chair at a desk. This being the case, we will not immediately employ a storage facility such as a queue, array, or linked list as a container for `Chair` objects. However, if we succeed in our committed relationship goals we may try to locate and place all chairs. In this case, we will store `Chair` objects into a queue, array, or linked list. With the latter implementation, this method will play a more vital role than in the case of our committed relationship goal.

Private Chair Attributes:

```
private int Color
```

`Color` is the RGB value of the sticker that we plan to assign to `Chair` objects. Using the camera and the color recognition subsystem of the robot, we can use color as a means of object recognition. We will define this color as a common RGB integer value for all `Chair` objects. Therefore, all `Chair` objects will have the same value for this attribute.

```
private Location XY
```

The instance of `Location` class gives a (X,Y,Z) coordinate for the `Chair` object in the room. It will be used to store the location a `Chair` occupies in the room. Several classes depend upon the `Location` class.

```
private bool Placed
```

This is a Boolean function to determine whether a `Chair` has been placed at a desk. Initially when a chair is found it is set to false. Once the chair is placed, it is set to true.

Desk

The Desk class is a representation of the objects in which instances of Chair will be placed. The Desk class consists of a constructor, setColor, setLocation, getColor, and getLocation methods. It also has the int color, Location XY, and bool isEmpty attributes.

Public Desk Methods:

```
public Desk::Desk()
```

The constructor initializes an instance of Desk to Color(-1), Location XY(-1,-1,-1), and bool isEmpty(false). XY must first instantiate an object of type Location, using its constructor for initialization of this object.

```
public void Desk::setColor(int color)
```

The setColor method defines the color that will be used to represent an instance of Desk. All Desk objects should have their color attribute set to the same value since we will be using a single color to describe objects of this type.

```
public void Desk::setLocation(Location XY)
```

Depending upon the implementation we get accomplished, this procedure will be different. In the committed relationships, the locations will be hard-coded, thus making this function obsolete. In the casual relationship, the setLocation method sets a Location object of the corresponding instance of Desk. We will assume, for the most part, that instances of Desk are static objects and will remain in the same location. This means we could store the location of all Desk objects in accordance with the room and never have to store them again. Therefore, we could start the robot and assume the location of Desk objects have remained constant.

```
public int Desk::getColor()
```

The getColor method returns the Color value of the corresponding instance of Desk. This attribute will be used to visually identify a Desk object.

```
public void Desk::getLocation(Location &Loc)
```

The getLocation method will pass a Desk Location back by reference. Our committed deliverable is to locate and place a single chair at a desk. This being the case, we will not immediately employ a storage facility such as a queue, array, or linked list as a container for Desk objects. However, if we succeed in our committed relationship goals we may try to locate and place all chairs. In this case, we will store Desk objects into a queue, array, or linked list. With the latter implementation, this method will play a more vital role than in the case of our committed relationship goal.

Private Desk Attributes:

```
private int Color
```

Color is the RGB value of the sticker that we plan to assign to Desk objects. Using the camera and the color recognition subsystem of the robot, we can use color as a means of object recognition. We will define this color as a common RGB integer value for all Desk objects. Therefore, all Desk objects will have the same value for this attribute.

```
private Location XY
```

The instance of Location class gives a (X,Y,Z) coordinate for the Desk object in the room. It will be used to store the location a Desk occupies in the room. Several classes depend upon the Location class.

```
private bool isEmpty
```

This is a Boolean function to determine whether a Chair object has been placed at an instance of Desk. Initially when a desk is found it is set to true. Once the chair is placed, it is set to false.

Error

The Error class is a general structure for reporting and logging errors that occur while the robot is running and trying to place chairs in their rightful Locations. It consists of a constructor, setError, and

getError methods. Its attributes consist of int errorNumber, Date errorDate, char picLocation[80], int errorDesc, and RandomObject* randObjPtr.

Public Error Methods:

```
public Error::Error()
```

The constructor will initialize errorNumber(-1), Date errorDate(-1, -1, -1,-1), and char picLocation(NULL). Class member errorDate must first instantiate an object of type Date, using its constructor for initialization of this object.

```
public void Error::setError(int errorNumber, Date errorDate, char picLocation[80], int errorDesc, RandomObject &ROP)
```

The setError function will set the error number, log the Date the error occurred, attempt to log a picture of the situation, give a descriptive error number for a pre-defined error state, and assign a pointer to a RandomObject if one was involved.

```
public void Error::getError(int &numberBuffer, Date &dateBuffer, char &locationBuffer,int &descBuffer, RandomObject *ROP)
```

The getError function returns by reference an errorNumber, errorDate, picLocation, errorDesc, and pointer to a RandomObject if one was involved.

Private Error Attributes:

```
private int errorNumber
```

This is the numerical value to describe the error. As development continues we will define error numbers to the many error states that may be encountered by the robot during operation. Example, error 404 chair not found.

```
private Date errorDate
```

This is the date that the error occurred. The Date object is defined in the Date class further in this document. This will play a key role in helping differential one error from another that may have occurred in a similar context.

```
private int errorDesc
```

If the error is one that has been acknowledged and prepared for it will be assigned a description to help remedy the problem.

```
private RandomObject* randObjPtr
```

If the error involves interaction with a random object, this attribute will be set to an address of an object of RandomObject type.

Wall

The Wall class is to aid us in mapping the room dynamically, detecting objects such as the fold up panels, which may or may not be in the robot's way each time it is operated. This class may or may not need to be used.

Public Wall Methods:

```
public Wall::Wall
```

The wall constructor will initialize Location beginPoint(-1,-1,-1), Location endPoint(-1,-1,-1), and float length(0).

```
public void Wall::setBeginPoint(Location beginPoint)
```

The setBeginPoint function is used to define the beginning of a wall. The beginning being relative to whichever side of the wall we define to be the beginning. An instance of class Location will be representing the (X,Y,Z) coordinates of the beginning point of the wall.

```
public void Wall::setEndPoint(Location endPoint)
```

The setEndPoint function is used to define the end of a wall. The end being relative to whichever side of the wall we define as the last point. An instance of class Location will be representing the (X,Y,Z) coordinates of the ending point of the wall.

```
public void Wall::setLength(float length)
```

The setLength function sets the length of the wall to the float value passed as a parameter.

```
public void Wall::getBeginPoint(Location &beginPoint)
```

The getBeginPoint function will pass a Wall beginPoint Location object back by reference.

```
public void Wall::getEndPoint(Location &endPoint)
```

The getEndPoint function will pass a Wall endPoint Location object back by reference.

```
public void Wall::getLength(float &length)
```

The getLength pass back the total wall length by reference. This will be the distance between the beginPoint and endpoint of the wall.

Private Wall Attributes:

```
private Location beginPoint
```

Since walls are basically line segments (2 connected points) this is one of the three attributes that are required. The first point of a found wall will be considered the beginPoint. It uses the Location class to describe both the beginPoint and endPoint

```
private Location endPoint
```

This is the sister point to the begin point. It has the same features as beginPoint only it is considered to be the termination point of the line, which denotes the end of a wall.

private float length

Length is the distance between the endPoint and beginPoint. As of this writing, it remains to be determined if this value will be measured or calculated. However, this could be the case for any of the attributes that is if you know two of the three attributes.

RandomObject

The RandomObject class is a representation of an object that the robot cannot understand with its learned capabilities. This object is neither a wall, desk, nor chair therefore is treated as a random object.

Public RandomObject Methods:

```
public void RandomObject ::setDOC(Date DOC)
```

The setDOC method is used to store the DOC (date of occurrence) a RandomObject was encountered. DOC is a private member of the RandomObject class.

```
public void RandomObject ::setPicLoc(char picLoc[])
```

The setPicLoc method accepts a character string parameter. This string is the fully qualified path name to the picture on the backing store. An example follows:

```
/home/rafs/RandomObject/randObj001.jpg
```

```
public void RandomObject ::getDOC(Date &DOC)
```

The getDOC method is used to retrieve the DOC (date of occurrence) a RandomObject was encountered.

```
public void RandomObject ::getPicLoc(char &picLoc)
```

The getPicLoc method passes a character string back by reference. This string is the fully qualified path name to the picture on the backing store. An example follows:

```
/home/rafs/RandomObject/randObj001.jpg
```


Private RandomObject Attributes:

```
private Date dateOfEncounter
```

The dateOfEncounter is the date the robot encountered a RandomObject. The Date class is described later in this document.

```
private char picLocation[80]
```

When an encounter occurs with a Random object that cause an error, a picture will be taken and stored at a specific path, such as "/home/rafs/RandomObject/randObj001.jpg". This string is the fully qualified path name to the picture on the backing store.

Location

The Location class is for describing an object's location somewhere in room 2029. It uses standard Cartesian coordinate system using floating-point numbers.

Public Location Methods:

```
public void setXYZ(float x, float y, float z)
```

The setXYZ method is used to set the (X,Y,Z) coordinates of a Location class object. The X,Y,Z values being float type private members of the Location class.

```
public void getXYZ(float &x, float &y, float &z)
```

The getXYZ method is used to retrieve the (X,Y,Z) coordinates of a Location class object. The X, Y, Z values being float type private members of the Location class.

Private Location Attributes:

```
private float xPt
```

This is the x coordinate of the location of a point somewhere in the room.

```
private float yPt
```

This is the y coordinate of the location of a point somewhere in the room.

```
private float zPt
```

This is the z coordinate of the location of a point somewhere in the room. This last attribute won't always be used in mapping the location of objects in the room but may be helpful for future use, especially if robots purchased in the future contain the ActivMedia Arm facility.

Date

This class is used for logging dates for events that occur.

Public Date Methods:

```
public void Date::setMonth(int month)
```

The setMonth method will set the Date class's private integer value Month attribute.

```
public void Date::setDay(int day)
```

The setDay method will set the day member of the invoking Date class object.

```
public void Date::setYear(int year)
```

The setYear method will set the year member of the invoking Date class object.

```
public void Date::getMonth(int month)
```

The getMonth method will pass back by reference the Date object's private member month.

```
public void Date::getDay(int day)
```

The getDay method will pass back by reference the Date object's private member day.

```
public void Date::getyear(int year)
```

The getYear method will pass back by reference the Date object's private member year.

```
public void Date::setTime(int time)
```

The setTime method is used to timestamp when an error, desk, chair, or RandomObject was encountered. The private time member is set to the passed parameter.

```
public void Date::getTime(int time)
```

The getTime method is used to retrieve a timestamp of an error, desk, chair, or RandomObject. This aids in adding a certain degree of uniqueness to each error.

Private Date Attributes:

```
private int month
```

The month of the year represented as an integer 1-12.

```
private int day
```

The day of the week the event occurs on represented 1-7.

```
private int year
```

The year the event occurs 0000-????.

```
private int time
```

The time attribute represents a timestamp value. This indicates when a desk, chair, RandomObject, or an error was encountered.

Classes for the Robot Aided Feng Shui Project

```
// About these classes: These are the base classes for the  
// implementation of some of the classes we intend to use for our  
// senior project. These classes all have to do with the  
// operation of ActivMedia robots and things they will encounter.  
// A brief description of each class and its attributes and functions  
// is described before the listing of each class.
```

```

// class: Chair
// description: The Chair class is a class of the very objects we will
// be moving back to their proper location at a desk. Chairs have
// several attributes as defined below
// attributes:
// color - Color is the color of the label that we plan to put on the
// Chair. Using the camera facility of the robot and the color
// recognition of the API we can use colors as a tool for object
// recognition.
// XY - The location class gives an (x,y,z) coordinate of the Chair
// in the room. The Location class is used by several classes. XY is
// the beginning point from where a Chair starts
// placed - This is a boolean function to determine whether or not a Chair
// has been placed at a desk. Initially when a chair is found it is
// set to false. Once the chair is placed, it is set to true.

```

```

class Chair
{
public:
    Chair();                //constructor
    void setColor(int color); //input
    void setLocation(Location XY); //input
    int getColor();         //output
    void getLocation(Location &loc); //output
private:
    int color; //doesn't matter what the color is, just as
              //long as it contrasts with the color of the Desk
    Location XY; //location of the chair within the room
    bool placed; //is the chair at the desk or not
};
//end of class Chair

```

```

// class: Desk
// description: The Desk class is a class of the objects where we will
// be moving back Chairs to their proper location. Desks have several
// attributes as defined below
// attributes:
// color - color is the color of the label that we plan to put on the
// Desk. Using the camera facility of the robot and the color
// recognition of the API we can use colors as a tool for object
// recognition.
// XY - The location class gives an (x,y,z) coordinate of the Desk
// in the room. The Location class is used by several classes. The
// Location is the finish or end point from where a Chair starts.
// is Empty - This is a boolean function to determine whether or not a

```

```
// chair has been placed at a desk. Initially when a Desk is found it is
// set to true. Once the chair is placed, it is set to false.
```

```
class Desk
{
public:
    Desk();                //constructor
    void setColor(int color);    //input
    void setLocation(Location XY); //input
    int getColor();            //output
    void getLocation(Location &Loc); //output
private:
    int color;    //doesn't matter what the color is, just as
                //long as it contrasts with the color of the Chair
    Location XY; //location of the desk within the room
    bool isEmpty; //is there a chair at the desk?
};
//end of class Desk
```

```
// class: Error
// description: The Error class is a general structure for reporting and
// logging errors that occur while the robot is running and trying to
// put Chairs back in their rightful Locations. Errors have several
// attributes as defined below.
// attributes:
// errorNumber - This is a number to log the error. For example, if
// the error to be logged is the 25th error in operation, this
// value will be set to 25.
// errorDate - This is the date that the error occurred. the Date object
// is defined in the Date class further in this document.
// picLocatoin - When an error occurs, the camera will take a picture of
// the error condition and store it in a folder. This attribute will
// store the location of the path where the file is located.
// i.e. //csfs2.siu.edu/sp/s02g2/picLog/pic25.jpg
// errorDesc - If the error is recognizable, it will be assigned a
// predetermined error number.
// randObjPtr - if the error involves interaction with a random object,
// this attribute will set a pointer to an object of the RandomObject
// type.
```

```
class Error
{
public:
    Error();                //constructor
    void setError(int errorNumber, //the sequential number of the error
                 Date errorDate, //the date of the error
```

```

        char picLoation[80],      //the location of the picture of the
                                //error
        int errorDesc,           //an error descriptor, i.e. error 123
        RandomObject &ROP);     //pointer to a RandomObject

    void getError(int &numberBuffer, //the sequential number of the error
                 Date &dateBuffer,  //the date of the error
                 char &locationBuffer, //the location of the picture of the
                                    //error
                 int &descBuffer,    //an error descriptor, i.e. error 123
                 RandomObject *ROP); //pointer to a RandomObject
private:
    int errorNumber;             //number to log the error
    Date errorDate;              //their may exist predefined date class
                                //for C++
    char picLocation[80];        //location of the picture of
                                //the error condition
                                //e.g., "usr/local/bin/errors/"
    int errorDesc;               //description of the error
    RandomObject* randObjPtr;
    //if the error involves encounters with a random
    //object, this pointer will point to that RandomObject.
};
//end of class Error

// class: Wall
// description: The Wall class is to help us with mapping the room
// dynamically
// with object such as the fold up screens that may or may not be in the
// robot's way every time it is operated. This class may or may not need
// to be used.
// The attributes of the Wall class are described below.
// attributes:
// beginPoint - since walls are basically line segments (2 connected
// points) this is one of the three attributes that it needs. The first point
// of a found wall will be considered the beginPoint. It uses the
// location class to describe the point.
// endPoint - This is the point that is the sister point to the begin point. It
// has the same features as beginPoint only it is considered to be the
// termination point of the line.
// length - Length is the distance between the endPoint and beginPoint.
// As of this writing, it remains to be determined if this value will be
// measured or calculated. However, this could be said of any of the
// three attributes as if you have two of the three attributes.

```

```

class Wall
{
public:
    Wall();
    void setBeginPoint(Location beginPoint); //input
    void setEndPoint(Location endPoint); //input
    void setLength(float length); //input
    void getBeginPoint(Location &beginPoint); //output
    void getEndPoint(Location &endPoint); //output
    void getLength(float &length); //output
private:
    Location beginPoint; //begin point
    Location endPoint; //end point
    float length; //distance between those two points
};
//end of class Wall

// class: RandomObject
// description: The RandomObject class is a class for objects that are
// encountered that cause the robot errors. The attributes are described
// below.
// dateOfEncounter - The Date of the encounter with the RandomObject.
// The Date type is described elsewhere in this document.
// picLocation - When an encounter occurs with a Random object that
// cause an error, a picture will be taken and stored at a specific path
// such as "C:\Windows\Temp\RandObj.jpg". This attribute is a
// character string for that path.

class RandomObject
{
public:
    void setDOC(Date DOC); //set the date of the encounter with
//the object
    void setPicLoc(char picLoc[]); //take a picture and the location is the
//path to it
    void getDOC(Date &DOC); //get the date of the encounter with
//the object
    void getPicLoc(char &picLoc); //take a picture and the location is the
//path to it
private:
    Date dateOfEncounter; //date of encounter with the random
//object
    char picLocation[80]; //location of the picture of the error
//condition
//e.g., "usr/local/bin/randObj"
}; //end of class RandomObject

```

```

// class: Location
// description: The Location class is for describing an object's location
// somewhere in the room. It uses standard Cartesian coordinate
// system using floating point numbers (not integers). The attributes for
// the Location class are described below.
// attributes:
// xPt - this is the x coordinate of the location of a point somewhere in
// the room.
// yPt - this is the y coordinate of the location of a point somewhere in
// the room.
// zPt - this is the z coordinate of the location of a point somewhere in
// the room. This last attribute really won't be used in mapping the
// location of things in the room but may be helpful for future use.

```

```

class Location
{
public:
    void setXYZ(float x, float y, float z);        //set the points
    void getXYZ(float &x, float &y, float &z);    //get the points
private:
    float xPt;        //x coordinate
    float yPt;        //y coordinate
    float zPt;        //z coordinate, not really used
};
//end of class Location

```

```

// class: Date
// description: This class is used for logging dates for events that occur.
// The attributes for the class are listed below.
// attributes:
// month - the month of the year represented as an integer 1-12.
// day - the day of the week the event occurs on represented 1-7.
// year - the year the event occurred.
// time - the time the event occurred.

```

```

class Date
{
public:
    void setMonth(int Month);    //input
    void setDay(int Day);        //input
    void setYear(int Year);       //input
    void setTime(int Time);      //input

    void getMonth(int &Month);    //output
    void getDay(int &Day);        //output

```



```

void getyear(int &Year);      //output
void getTime(int &Time);     //output

private:
    int month;               //the month of the year
    int day;                 //the day of the month
    int year;                //the year that date occurs
    int time;                //the time of the date.
};
//end class Date

```

4.4 Networking and Database Information

Network Information

The ActivMedia Robot will be controlled over the SIUE School Of Engineering Windows NT network (which from here on out will be referred to as SOENT) via a TCP/IP socket connection using a wireless Ethernet adapter made by Lucent Technologies. This hardware/software combination allows us to connect to the robot via any of the PC's in the Engineering Building. However there is one limitation. The Ethernet adapter has two ends to it: one for the robot and one that plugs into the standard network cabling via the network jacks spread throughout the building. The robot must be sufficiently close to the other wireless adapter to promote reliable communication and fewer network errors. The team has found that this is an obstacle that can be overcome by moving the second wireless adapter to whatever room the robot is operating in.

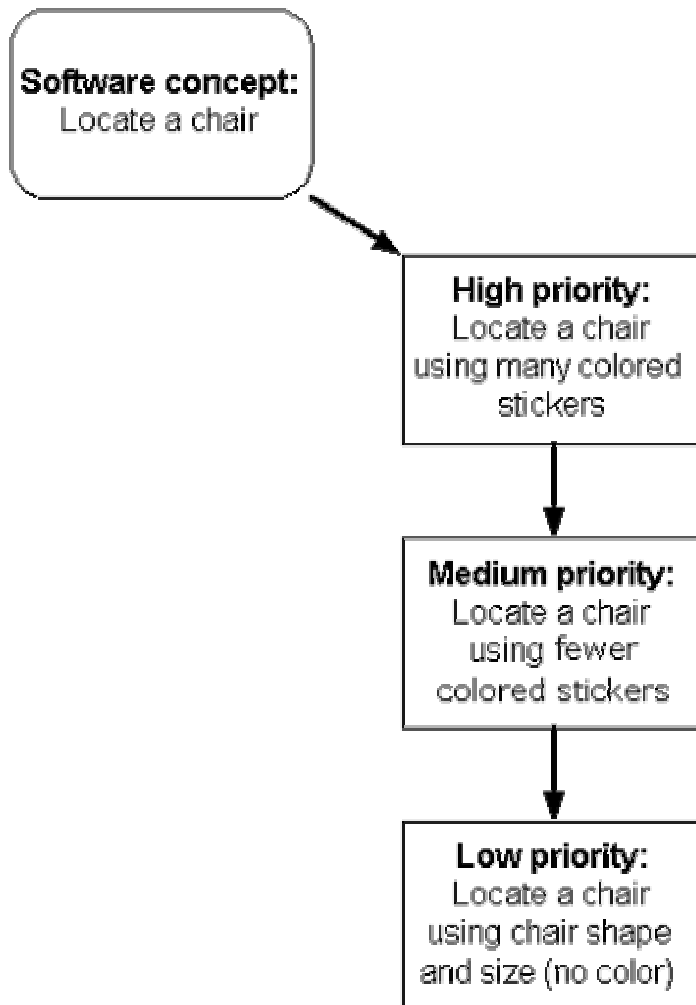
It has become one of our long-term goals to make the Robot Aided Feng Shui software accessible via the World Wide Web. This seems achievable since the World Wide Web is just a "giant" IP network, similar to the SOENT network that the robot currently runs on. Part of our long-term goals for this involves using a web browser to "drive" the robot from a remote (off campus) location.

Database Information

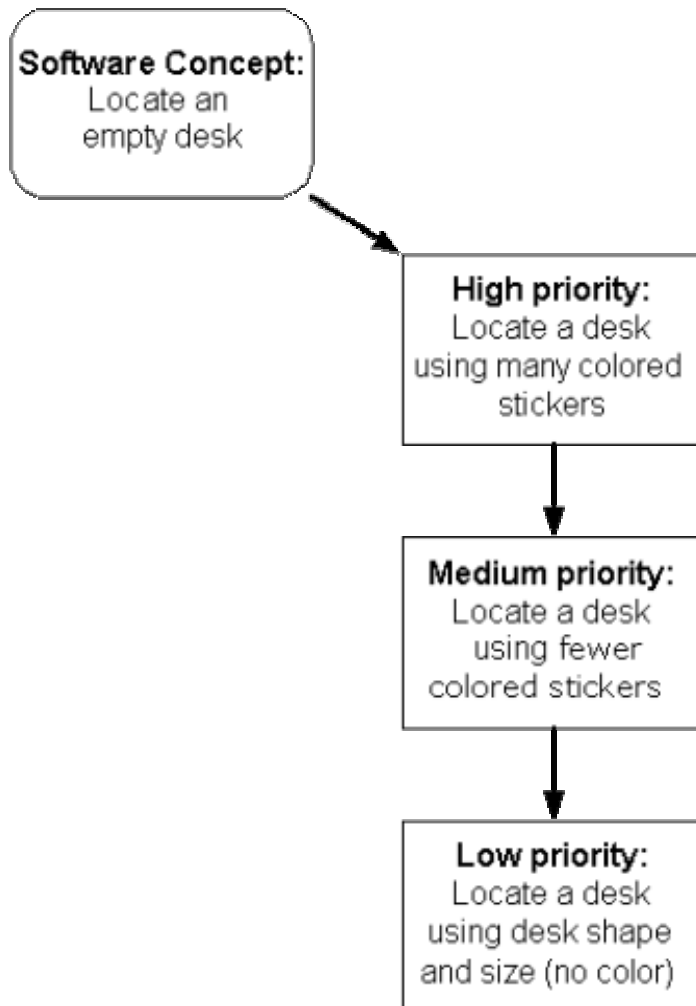
As the design of this project currently stands, there is not a database implementation. However, for future growth, we have planned to use a database for logging and keeping track of errors and error statistics. The type of database we will use is the relational style database. This style of database has multiple tables with related entries. The database

we would like to use would most likely be an SQL compatible database that is also easily compatible with UNIX/Linux systems. This leads us to believe that an ORACLE database written in the C or C++ programming language would be most helpful to us and most likely to stay with the current trend of ORACLE and SQL being leaders in database technology. We reserve the right to change our future implementations of the database based upon current technology, costs, and the experience of the team members.

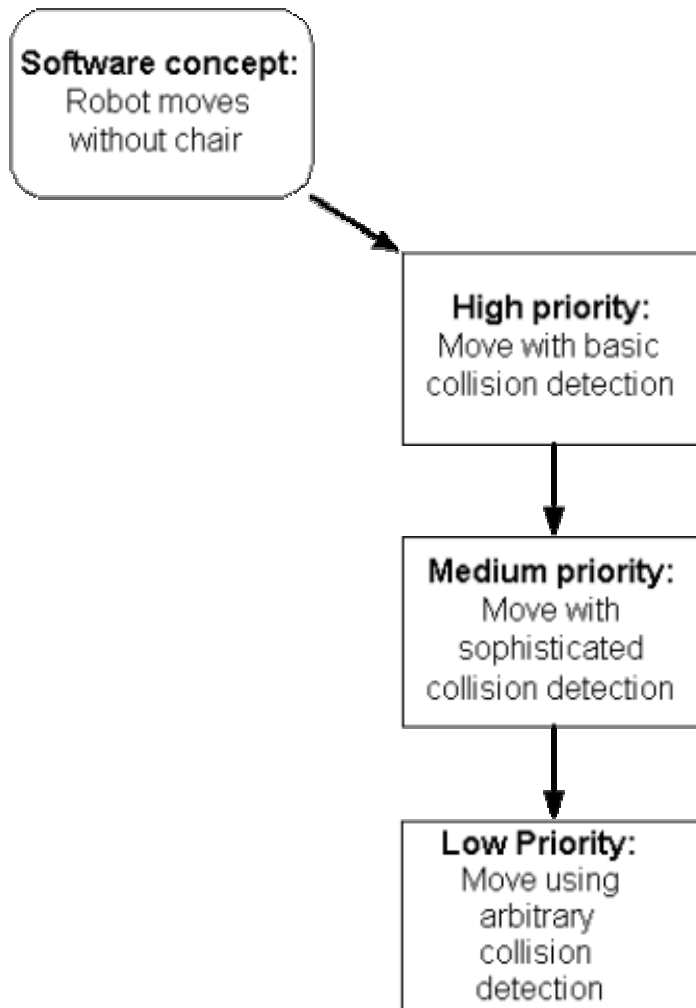
4.5 Lifecycle Model



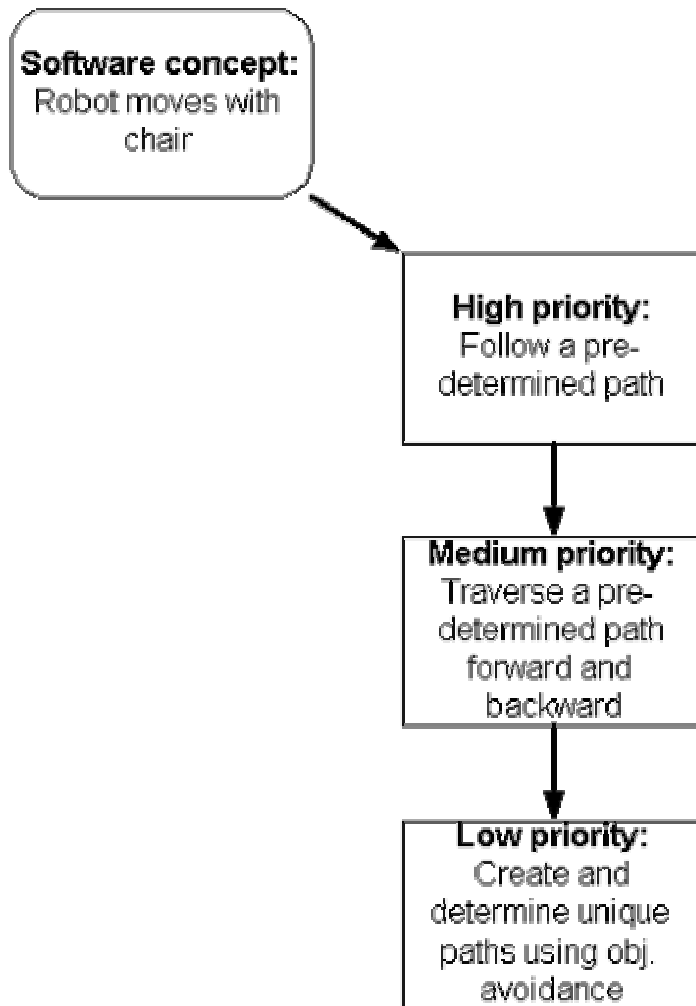
This diagram represents the approach we will be taking to module development with regards to chair objects. We will first attempt to locate chairs via a highly concentrated amount of colored stickers being placed at strategic positions on the chair. Successive attempts will involve fewer colored stickers eventually eliminating the use of color all together.



This diagram represents the approach we will be taking to module development with regards to desk objects. We will first hard code the position of the desks. Successive attempts will involve a few colored stickers, eventually eliminating the use of color all together.



This diagram represents the approach we will be taking to module development with regards to robot movement without a chair. Initially we will take advantage of the robot's built-in object avoidance features. As time permits, we would like to engineer further, more sophisticated techniques, to avoid collisions with static objects. Finally, we would attempt to avoid collisions with all sorts of dynamic objects that may occur in EB2029.



This diagram represents the approach we will be taking to module development with regards to robot movement while holding a chair. Our first attempt will utilize a predetermined path that will be clear of obstructions and navigated in the forward (pushing) direction only. Following efforts will have the robot operate in both a forward (pushing) and backward (pulling) fashion. Final attempts would be concentrated on creating unique paths using sophisticated object avoidance techniques. This could involve creating the path while moving a chair, rather than beforehand.

Design Conclusions

Our design model, design to schedule, lends itself quite nicely to our project in particular. Our minimal requirements are to locate and place a single chair at an empty desk.

Our initial development will be geared toward locating a chair using several colored stickers with subsequent phases reducing the use of color until chair recognition is solely based on its physical characteristics. Given our time constraints and learning curve, we realize that it is highly likely that we would not be able to implement all phases in the allotted time. However, our design takes this into account leaving refinement as less of a priority than finishing the minimal client requirements.

All other software concepts take much the same approach as dealing with recognition of a chair; refinements are something to strive for, but not essential, to successful completion of our project.

Our design will leave us with a fully deliverable software package with at least the minimal client requirements by our December 2002 deadline. As time permits, subsequent phases of refinement on software concepts will be implemented in order to deliver the most robust piece of software possible.

4.6 Test Plan

A significant amount of testing will be done to each software modules that is constructed for our project. This module breakdown is attached at the end of this document and shows our current organization plan. Each module will be tested individually using stub data if needed to ensure it behaves as intended.

The Pioneer simulator provided with Saphira 8.1 adds a valuable tool to our testing process. The simulator is capable of emulating each sensory device providing stub data that would be expected from each sensory subsystem. This being the case, software modules will be testable without the robot being available. This provides us with an extremely flexible environment in which we can ensure the correctness of our code modules.

Per Module Testing

Movement

Two types of tests will be conducted on the movement subsystem, both with and without a chair.

Without Chair

This module will be expected to handle the basic movement of the robot in the EB2029 environment. The test could be conducted using the mapping features of the Pioneer simulator or in the live physical environment. Live testing would produce more meaningful results since dynamic objects may be present, therefore testing the robustness of our movement handling. Using the mapping features of the simulator would test the handling of static objects and room navigation in general. Overall, both testing methods would provide us with a good idea of how well we are handling the basic navigation aspects of EB2029.

With Chair

This module will be expected to handle the basic movement of the robot in the EB2029 environment while navigating with a chair. This test will not be able to be conducted via the simulator because of the lack of functions emulating an object being held. As our research continues we may find a way to emulate this state of movement. To ensure appropriate behavior, a greater effort will be exerted in testing this module in the live physical environment of EB2029, rather than attempting to emulate this via the simulator.

Object Recognition

Key elements of object recognition will require aggressive testing to ensure a collision will not occur during the ordinary operation cycle.

Color Training

The robot will be able to distinguish object from each other via color training. This software module can be tested on series of sample photos taken from the robot's mounted camera. This testing can be done without the need of the robot, therefore making it possible to test on an ordinary PC.

Laser/Sonar

These subsystems may come into use as our obstacle avoidance algorithms mature. The Pioneer simulator is able to emulate these devices - feeding stub data that would be expected from a certain environment. To ensure the accuracy of our tests, they will also be carried out with the robot in the live EB 2029 environment.

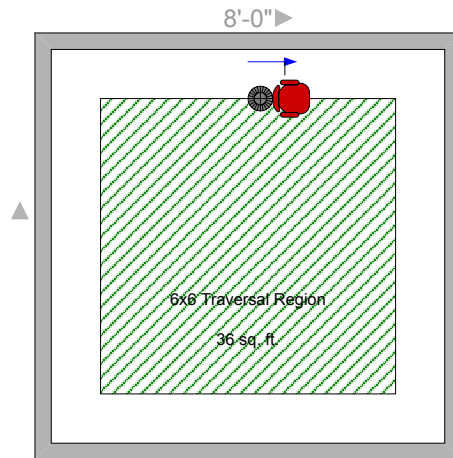
Camera

This will be the primary tool for recognizing empty desks and chairs. The camera will be able to recognize a certain color as being an empty desk or a chair. We will recognize any other object as being a type defined as "other." Basically, we will simply ignore anything that is not an empty desk or a chair.

Chair Handling

The primary module for Chair Handling will be Moving with a chair. That will take care of any collision detection, along with knowing when a chair is at a desk. Chair Handling will also have to deal with gripping a chair, which will be the first thing we will test in this module. From that point, we will combine the gripping a chair with the already-written module, Moving with a chair.

4.7 Project Prototype



Class Description

In order to complete our project prototype, our group had to work in an expedient fashion, quickly learning the new tools and environments used in our project. As a result of a hardware malfunction on the robots, we had significantly less time to develop our prototype code than expected. The Pioneer2 robot allocated for our senior project has been mailed back to ActivMedia and is pending service from their expert staff.

An attempt was made to complete this prototype despite the problems with the equipment, and significant progress was made. As it stands now, our prototype enables the robot to complete a 6-foot square in both a clockwise and counterclockwise direction. The robot maintains the grip of the chair through the entire process. However, it does not detect the loss of grip on the chair. The following breakdown of classes involved in our prototype will highlight their features and contribution to the entire prototype.

- *SfRafsGripperOpenAction*

This simple class allows us to invoke this action manually from the Saphira interaction window or call this function from a Colbert script. Its sole purpose is to open the gripper to the widest possible state preparing it for gripping a chair.

- *SfRafsGripperCloseAction*

This simple class allows us to invoke this action manually from the Saphira interaction window or call this function from a Colbert script. Its sole purpose is to close the gripper to the narrowest possible state therefore gripping a properly positioned chair.

- *SfRafsGripperRaiseLowerAction*

This class allows us to either raise or lower the position of the gripper. This action is necessary to ensure us the best position for gripping a chair. The parameters to the invoking function define whether the gripper is moving up or down and for what duration it should move. We would have liked to have been able to specify an exact position, however the only metric of movement is duration. Our static values for this function are 0 to move down and 1 to move up. To position the gripper at the appropriate height we lowered the gripper to the lowest possible position, *RafsGripperRaiseLower(0,0)*. Then we would raise the gripper upwards for 135 milliseconds. *RafsGripperRaiseLower(1,135)*

- *SfRafsSquareClockWiseAction*
SfRafsSquareCounterClockWiseAction

This class is the main code to our (counter)clockwise square movement action. The parameter to the invoking function accepts an integer which defines the length of one side of the total square wishing to be traversed. This parameter is to be sent in millimeters. Therefore the value 1829 was sent to achieve a 6 foot square traversal.

Code Structure

Our source code was based of a simple example that came packaged with the Saphira 8.1 API. We utilized this example because it contained example code for controlling robot movement and defining actions for use in Colbert within the Saphira GUI interaction window. There are some vital functions that define the fashion in which this library is loaded and accessed.

The class method *fire* is the main loop, so to speak. This method is executed continuously until it returns a null value. As long as a valid reference to a desired action is returned, this function will continue looping.

The class methods *invoke* and *sfLoadInit* define the interfaces to the Colbert scripting language. These functions enable us to call the functions manually via the Saphira GUI interaction window or via a Colbert script.

The remaining class methods have sufficient in-line comments to clarify their purpose and usage.

Proof of Concept

In order to provide proof that our prototype works as suggested, we contemplated several ways to present our prototype to the customer. Since the robot is unavailable for a formal presentation, we decided to rely on the Pioneer2 simulator and a videotaped session of the prototype taken when the robot was still available. While it is difficult to rely on the simulator for all testing, we hope it will be sufficient for this part of the project.

Example Code

An example program from our prototype, following the procedure in the Programming Standard section is provided below. The example used is a module from our prototype that gets the robot to move in a user-defined square. It works (for the prototype) with a couple other modules that move the gripper. The *SfRafsSquareCounterClockWise* is as follows:

```
// RAFS - Robot Aided FengShui
// Souther Illinois University at Edwardsville
// Department of Computer Science

// file: rafsSquareClockWise.cpp
// author: Peter Motykowski

#include "Saphira.h"

#include <export.h>
#include <math.h>

// class: SfRafsSquareClockWiseAction

// description: Defines a Colbert action which has the robot traverse a n-sized
//              square. This class takes one parameter, the size of one side of
//              square in millimeters.
```

```

// attributes:
// Desired - This class member is not used in this class, however
//            it is left to provide the necessary parameters to *fire().

// Distance - This class member defines the size of sides of the square.

// Gone -     This class member keeps track of how far has been traveled.

// Direction - This class member keeps track of the direction the robot will
//            be heading next.

// ax,ay -    These class members are used to keep track of the robots current
//            position.

// ExitProgram-This class member is used to determine when the program should exit.

```

```

class SfRafsSquareClockWiseAction : public ArAction

```

```

{
public:

    SFEXPORT SfRafsSquareClockWiseAction(
        int distance
    );

    virtual ~SfRafsSquareClockWiseAction() {};

    SFEXPORT virtual ArActionDesired *fire(
        ArActionDesired currentDesired
    );

    SFEXPORT void reset() {
        Gone = 0;

        ax = SfROBOT->getX();

        ay = SfROBOT->getY();
    }

    static SfRafsSquareClockWiseAction *invoke(
        int distance
    ); // interface to Colbert

```

```

protected:

```

```

    ArActionDesired Desired;

    int Distance; // parameters to the action

    int Gone; // how far we've gone

```

```

    int Direction;

    double ax;
    double ay;           // old robot position

    bool ExitProgram;
};

// function name: SfRafsSquareCounterClockWiseAction
// parameters-in: distance - length of one side of the square
// parameters-out: none
// parameters-in/out: none
// returns: void
// description of function: initializes all the variables needed later in the class
// notes: constructor

SFEXPORT
SfRafsSquareClockWiseAction::SfRafsSquareClockWiseAction(int distance)
: ArAction("RafsSquareClockWise")
{
    Distance = distance;

    Direction = 360;

    reset();

    ExitProgram = 0;
}

// function name: fire
// parameters-in: d - a pointer of type ArActionDesired
// parameters-out: none
// parameters-in/out: none
// returns: returns an ArActionDesired pointer, containing what the
// action wants to do
// description of function: This is the main part of the program. This function
// runs the distance initiated in the constructor, then turns left, and runs the
// distance again, continuing the process until a square is traversed.
// notes: none

SFEXPORT ArActionDesired *
SfRafsSquareClockWiseAction::fire(ArActionDesired d)
{
    Desired.reset();           // reset the actionDesired (must be done)

    double dx = ax - SfROBOT->getX();
    // check the distance to be traveled
    double dy = ay - SfROBOT->getY();

    ax = SfROBOT->getX();

```

```

//set new values
ay = SfROBOT->getY();

int ds = (int)sqrt(dx*dx + dy*dy);

Gone += (int)ds;

//sfMessage("Running RafsSquareClockWise, gone %d", myGone);

if (ExitProgram)
{
    sfMessage("Shutting Down!");

    deactivate();

    return NULL;
}

if (Gone >= Distance)
{
    if(Direction > 0)
        Direction -= 90;

    else
    {
        ExitProgram = 1;

        Desired.setVel(0);

        sfMessage("Finishing RafsSquareClockWise");

        return &Desired;
    }

    sfMessage("Running RafsSquareClockWise, direction %d", Direction);

    Desired.setHeading(Direction);

    sfMessage("Turn Right");

    Gone = 0;
}

else
    Desired.setVel(200);           // moderate speed

ExitProgram = 0;

return &Desired;                 // return the desired controls
}

```

```

// function name: invoke
// parameters-in: distance - length of one side of the square
// parameters-out: none
// parameters-in/out: none
// returns: returns a new SfRafsSquareCounterClockWiseAction object
// description of function: This static function returns a behavioral action object,
// with arguments that can be set from Colbert
// notes: interface to Colbert

SfRafsSquareClockWiseAction *
SfRafsSquareClockWiseAction::invoke(int distance)
{
    return new SfRafsSquareClockWiseAction(distance);
}

// function name: sfLoadInit
// parameters-in: none
// parameters-out: none
// parameters-in/out: none
// returns: void
// description of function: Defines the interface to Colbert
// notes: none

SFEXPORT void                // define interface to Colbert
sfLoadInit ()
{
    sfAddEvalAction("RafsSquareClockWise",
                    (void *)SfRafsSquareClockWiseAction::invoke, 1, sfINT);
}

```

syntax highlighted by [Code2HTML](http://Code2HTML.com), v. 0.8.11

5. Attachments

Attachments are on the next few pages, in the following order, with one diagram on a page:

- Module Diagram
- State Diagram
- Sequence Diagram

Note: The actual attachments are located in modules.gif, NewStateChart.gif, and sequence.gif, respectively.