April 2008

META-REASONING DIAGNOSIS AGENT

Elijah D. Forbes-Summers
Worcester Polytechnic Institute

Michael Longqiang Zhang
Worcester Polytechnic Institute

Project Number: CS-DCB-07N1

META-REASONING DIAGNOSIS AGENT

A Major Qualifying Project Report:

submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

Michael Zhang **Elijah Forbes-Summers**

Date: April 24, 2008

Approved:

Professor David C. Brown, Advisor

1. AI
2. diagnosis
3. agent

Abstract

This project investigates the feasibility of using a multiple-reasoner diagnosis agent to perform robot-to-robot repair during multi-robot NASA missions. We designed and built a proof-of-concept agent that simulates interfacing with another robot in order to perform diagnosis of apparent defects and attempt mitigation. The agent uses multiple types of reasoning and is capable of selecting the most appropriate type for the task at hand.

Acknowledgements

We would like to thank our advisor Professor David C. Brown for his guidance and patience. This project would not have been possible without him. We would also like to thank our NASA advisor Walt Truszkowski and also Brian Roberts and Paul Kim, both from NASA. They were all extremely helpful in guiding and commenting on our work.

Contents

List of Figures

1. Introduction

NASA uses robots for many of its space missions. The downside to unmanned missions is that when there are problems no one is there to fix them. However, on missions with more than one robot the robots could help each other. Our advisor at the NASA Goddard Space Flight Center, Walt Truszkowski, is one of the engineers working to make this type of robotic cooperation possible.

For robots to be able to help each other they need to access information about the other robots and be able to perform operations similar to those a human technician would perform. To allow these types of interaction a specialized hardware interface was conceptualized that all robots will be equipped with (Dearden, 2007). However, this is only half the problem. The robots also need to be “smart” enough to understand the information they can retrieve from other robots so that diagnosis can be carried out and the proper mitigation selected for the problem found.

Our task was to design and build a prototype version of this software. The software would have to reasonably simulate the function of a “Doctor robot” retrieving information from, and attempting to diagnose and treat, a “Patient robot”. It would also have to demonstrate this functionality visually and in an easily presentable manner.

Any terms unfamiliar to the reader (either of this report or some material cited by it) can likely be found with definitions in the Glossary.

1.1 Motivation

The robots that NASA sends on space missions are very expensive. Because of their cost it is important that a robot does not fail before the mission is completed. Engineers do not have physical access to the robots once they are launched so reliability is a primary concern during the design of these robots. Reliability has been increased by building redundantly; these robots usually contain duplicate components and have various backup methods in the case of failure. Additionally, the robots can perform a variety of self-checks to identify problems. The strategy for future lunar missions will be to build redundancy into not only the individual robot but also into a group of robots. A group of robots is more reliable than a single robot because robots in a group can help another robot that, on its own, would normally be unsalvageable or have reduced functionality.

In order for the robots to perform this “helpful” interaction each robot needs to be able to interface with any other. To provide this interface, NASA has developed the concept of a Diagnostic and Power Port (DPP), which will provide electrical power and communication between robots using direct connection (Dearden, 2007).

Through the DPP, a robot can access the status and internal data of another robot, as well as issue commands to it. This allows one robot to attempt to determine the cause of another robot’s failure and mitigate the problem.

1.2 Domain

NASA’s proposed missions to the moon call for multiple robotic rovers to be present on the lunar surface. They will be required to function mostly autonomously, including the ability to solve potential problems that may arise. The goal is to have astronauts living on the moon with robot assistance. This would be preceded by missions of semi-autonomous rovers to the moon. The rovers will prepare for the arrival of astronauts by preparing housing or performing mining operations, etc.

The domain of our project consists of NASA's future lunar rover missions. Although our project will be used solely for the evaluation of this approach in NASA's lunar rover context, it could also be applied to other areas. This project's underlying approach of using a high level reasoner to select from different reasoning strategies could be used in countless other domains where autonomous reasoning is required. It allows intelligent tradeoffs between reasoning techniques that would otherwise have to be selected by humans. The fact that much of our project is constructed in a modular way also makes it applicable to other domains without much reconfiguration.

1.3 Goals and Objectives

Our project is to create a software agent that uses soft computing techniques. It will have multiple diagnostic modules, each with different reasoning techniques used to perform diagnosis of a robot, and a meta-reasoner capable of determining which reasoner is appropriate, given a set of symptoms.

It will be a proof-of-concept implementation with the purpose of demonstrating the plausibility of key ideas, and provoking further thought and interest. This proof-of-concept should provide very shallow but wide coverage of functionality. We will demonstrate its functionality in a simulated situation through a GUI system that allows the user to specify a problem, and watch our agent perform its task.

We will implement three different reasoners for diagnosis: Case-Based Reasoning, Fuzzy Rule-Based Reasoning, and Neural Networks.

Our agent will have various levels of intelligence. The first level of intelligence involves reflexive actions in response to symptoms. The second level has our agent act as a communication relay so that humans can do the reasoning and issue commands (this is when it *decides* that it can’t solve the problem). The third level of reasoning involves autonomous diagnosis and treatment.

Our agent will include a basic implementation of some of the related components necessary for processing beyond the diagnostic reasoning, such as: mitigation reasoning, planning, scheduling, and execution of the treatment. These aspects are not the focus of the project, but a basic implementation will be developed for purposes of the proof-of-concept. The entire system will be designed modularly. If a more complex version of a module is developed, it can be quickly and easily ‘plugged-in’.

2. Background

In this section we present a brief overview of topics relevant to our work. We include an introduction to the reasoning techniques used by our system as well as “agent communication language” which inspired our own method for handling communication between robotic rovers (simulating the DPP).

2.1 Agent Communication Language

Agent Communication Languages (ACL) allow agents to communicate with each other (Woolridge, 2002). They can share knowledge, request information, request actions to be performed by another agent. They are based on the theory of “Speech Acts” that says that certain types of utterances have similar properties to physical actions (that they have a definite influence on the physical world). A system was built by Cohen and Perrault that allowed planning systems to reason about speech acts (Woolridge, 2002). Their system used preconditions and postconditions associated with various speech acts to indicate the requirements for uttering a speech act and the expected outcome of doing so. By connecting speech acts with models of data and agent knowledge a model of dialog between agents can be created. Common languages include the Knowledge Query and Manipulation Language (KQML) and the FIPA ACL, created by the Foundation for Intelligent Agents (Wikipedia, 2007).

In our project we will not be creating a full blown implementation of an ACL such as KQML or FIPA ACL, but will draw on these implementations and their guiding principles in order to create a simple ACL for communication between our diagnosing agent and the patient robot.

2.2 Case-Based Reasoners

Case-Based Reasoning is an AI technique that is in many ways similar to how humans solve common problems. There are four key steps in the Case-Based Reasoning approach to a problem (Kolodner 1993). First the Case-Based Reasoner must identify the problem at hand in terms it understands. Second the reasoner uses a store of previous problems for which it has solutions to map the new problem to the most similar previously known problem. Third the reasoner adapts the solution for the similar problem until it deems it a viable solution for the new problem. Finally the reasoner applies the solution to the new problem and stores the problem-solution pair for later use. Storing successfully applied solutions allows the reasoner to "learn" by expanding its solution space. Next time a problem is encountered the reasoner can apply a newly learned solution that may be a closer fit to the new problem than any other.

That is the simplest view of a Case-Based Reasoner. There is, of course, considerable complexity involved with each step of the process. There are problems associated with how to determine the similarity between problems, how to determine what types and magnitudes of adaptation to make to a solution so that it better fits the current problem, as well as how to determine the usefulness of a solution when deciding to store it for later use.

We will be using an off-the-shelf system so many of these problems will be partially handled by the tool.

2.3 Neural Networks

Artificial Neural Networks, commonly referred to simply as neural networks, are a statistical method of learning. ANNs are inspired by the network of neurons that make up the human brain. Neurons are cells whose primary function is the processing of electrical signals. A neuron takes in signals and produces a signal. Through an interconnected network of these neurons, the simple behavior of a single neuron interacts dynamically with other neurons to create the complex behavior of a network. Some of the major features of neural networks are their ability to handle noisy input, their potential for distributed processing, and their ability to be trained on sample data. They are typically used for pattern recognition, classification, and regression analysis.

Instead of neurons, neural networks have nodes. Multiple nodes are connected by links. Links are directional, allowing signals to pass only in the direction of the link. The link also has a weight, which modulates the signals that pass through it.

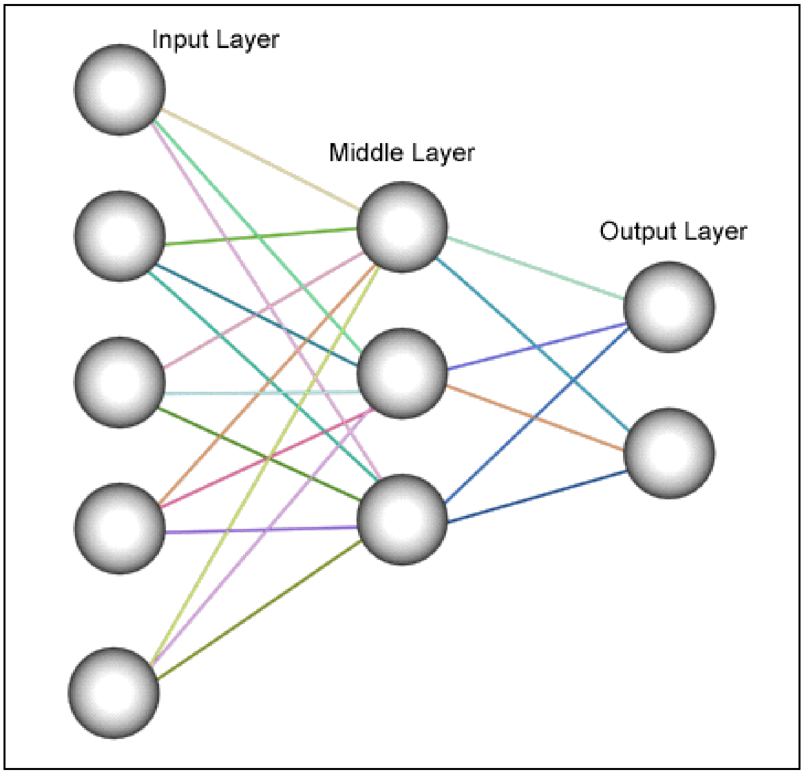


Figure 1 : A basic neural network structure with an input, hidden, and output layer (Kunzle, 2008).

A basic network of nodes, as shown in Figure 1, takes input from one or more nodes. These input nodes, are connected through links to a network of nodes which ultimately link to one or more nodes which represents the output. The inputs to a node consist of links to the node and a bias weight value. Each node takes the weighted sum of the signals from nodes that link to it, and applies an activation function to it. An activation function is simply a nonlinear function that, in combination with the bias weight, determines whether the input surpasses a threshold, thus activating the node.

With the appropriate weights, a node can be used to represent the basic logic gates: AND, OR, and NOT. This means that a network of neurons can be used to represent any complex Boolean function.

There are two major types of neural networks: feed-forward and recurrent. Feed forward networks are acyclic whereas recurrent networks are cyclic. Recurrent networks add the ability to have short-term memory. Since we do not have any need for short-term memory, our usage of and research on neural networks will be limited to feed-forward networks. From here onwards, the use of the term neural network will imply a feed-forward network.

Neural networks have layers of nodes. Each layer of nodes only receives input from the previous layer. Each layer also only links to nodes in the next layer. Single-layer networks are also referred to as perceptions. Perceptrons are limited to linearly separable functions, which makes them rather limited. Multi-layer networks are more powerful in that they can represent a larger set of Booleans functions.

Learning in neural networks is achieved by changing the weights of links. Since a neural network can be seen as a function of its inputs, by modifying the weights, we modify the function. The goal is to modify it such that given a certain set of inputs, we get the correct output. In the case of pattern recognition problems, the correct output would be the correct classification of the input pattern.

In a multi-layer network, learning can be done by back-propagation. Back-propagation is done by finding the error in an output node, and assigning a portion of the error to each of the input nodes linking to the output node based upon the weight of contribution of the input node. The links then update their weights and the error is propagated back another layer until it reaches the first hidden layer.

In order to apply a neural network, it needs to be trained before it is placed in the actual environment. That means that a training set of data needs to be available. Also, in order to learn, a neural network needs to compare its output with the correct answer. This information will only be available to our agent when either a human or another reasoner produces the correct output.

2.4 Rule-Based Systems

Intelligent behavior in humans is assumed to be at least partially rule-based in nature (Jackson, 1999). That is, humans usually follow a set of rules for determining how to act; even if they don't know exactly what the rules are. Based on this assumption there are a class of agents which use *production rules* in order to achieve intelligent behavior. The rules used by these systems consist of antecedents and consequents: when the antecedents of a rule are satisfied, its consequents are executed. Consequents can either provide additional knowledge that acts as an antecedent for some other rule, or can represent an action to be executed by the system.

The rules used by these systems need a way to represent information and their knowledge about the domain they are working in. A standard vocabulary for representing this information consists of: a set of object names, a set of attributes which name properties of objects, and a set of values that properties can have (Jackson, 1999).

Rule-Based Systems have a set of rules, a rule interpreter that finds rules to use based on the present conditions, and a working memory that stores the present conditions and any goals (Jackson, 1999). Rule interpreters operate over rules in two different ways: forward and backward chaining. Forward chaining is the more intuitive process. The rule interpreter looks for rule antecedents that match the present conditions and triggers the rule consequents. It proceeds in this manner until a conclusion is reached. Rule interpreters may also be designed to perform backward chaining. In backward chaining a final conclusion is hypothesized and the rule interpreter looks for rule consequents that would generate it. It then checks those rules to see if their antecedents can be triggered by any other rules' consequents. The interpreter continues until it finds rules that can be triggered based on the initial conditions alone.

2.5 Fuzzy Systems and Rules

Many human concepts are not sharply defined and have some degree of "fuzziness" in their definition. Examples of this are classifications such as "cold", "fast", or "near". This poses a problem for any tool that reasons using concepts such as these, especially when it has to translate discrete data into a fuzzy concept. Is 50°F cold? Where do we make the cutoff between warm and cold?

Fuzzy sets allow us to more easily handle this problem. The elements of normal sets ("crisp" sets) can either belong entirely or not at all, but fuzzy sets can handle degrees of membership. A crisp set called "cold" may include 50°F and therefore exclude it entirely from the set "warm". In the fuzzy version, 50°F may be a member of the cold set to degree 0.6 and also a member of the warm set to degree 0.4. This method of "partial" membership is more intuitive to human understanding of how membership in sets such as “cold” is determined. Allowing partial membership means that better comparisons can be made between elements; even 40°F is considered warm when compared to -10°F.

We can perform logical operations on fuzzy sets in a similar way that we do with crisp sets. For example, we can take the negation of whether 50°F is a member of the set “cold” and the answer is 0.4 (i.e., 1 – COLD(50°F)). We can also handle conjunction and disjunction, albeit in a slightly different way than normal logic. Lets say we had classified a certain comet as being a member of cold to degree 0.9 and of near, 0.4 (farther than the moon, but closer than Pluto). If we wanted to know whether the comet was cold *and* near we would find the minimum value of its individual memberships. So the comet is cold and near to degree 0.4. If we wanted to find if the comet was cold *or* near we would take the maximum membership, that is, 0.9.

Seemingly strange things happen when we try to find things like the conjunction of whether the comet is comet is cold and the negation of whether it is cold. In normal logic this would be zero (how could it both be and not be cold?), but in fuzzy logic the answer is 0.1. Because there is some uncertainty as to whether the comet is cold the answer to the problem of whether it is cold and not cold can be interpreted as: “how much does the comet belong to the set of *medium* cold things” (both cold and not cold)? We see that this is actually similar to the intuitive way people would answer this question.

3. Methodology

3.1 Schedule

Our project ran for three terms (A-C) from August 23, 2007 to February 28, 2008. We planned to do the requirements analysis and most of the design during A term so that we could do implementation during B term. During C term, we would do testing, and any final refinements necessary, and complete our project report document. We planned to work on our project report document over the course of all three terms.

3.2 Meetings

We held weekly meetings with Professor Brown. The purpose of these meetings was to present the work that we had done over the week, get any feedback about that work, and discuss the direction of our work for the upcoming week.

We will held weekly teleconferences with Walt Truszkowski, NASA GSFC, during which we discussed the requirements (during A term) of the project along with getting feedback on our work (B and C term). Some of the weeks, we were joined by Brian Roberts or Paul Kim, also of NASA, who provided us with additional specific information.

3.3 Development Tools

NASA did not provide us with any specific language requirements, but mentioned that some previous projects had used Java. We subsequently decided to use Java for our implementation of the project. We used Eclipse (http://www.eclipse.org/) for our development environment. Eclipse is an open source development platform popular for Java development. It provides excellent integration with Sourceforge (http://sourceforge.net/) source code versioning management. For source code version control, we used the WPI sourceforge. In addition, our rule based expert system package integrates with Eclipse as well.

3.4 Reasoner Tools

3.4.1 Java Object-Oriented Neural Engine

JOONE is a free Neural Network Framework that is used to implement the Neural Network Reasoner (Marrone, 2007). JOONE supports feed forward, recursive, time delay, back propagating, and resilient back propagating neural networks. It has a modular design, and consequently allows networks created and trained in a JOONE environment to be used in any other JOONE environment. This allows for the transference of reasoning knowledge between all agents using JOONE for neural networks. JOONE is also highly-scalable, and supports distributed processing as well. While most of these features are not applicable to the scope of our project, it will leave room for further expansion if desired.

JOONE is well documented and current. JOONE is available under the Lesser General Public License, and is free for all use.

3.4.2 Java Expert Shell System

JESS is a rule engine based upon the JSR94, the Standard Java Rule Engine API, that uses an enhanced version of the classic Rete algorithm for rule matching (Friedman-Hill, 2006). It is well documented and has an active community of users. It also has an Eclipse Plug-in, which integrates well with our Eclipse development environment. JESS also supports XML-based rule definition through its custom XML rule language called JessML. JESS is much faster than a naïve implementation because it uses Rete. However, the speed comes at a tradeoff between speed and memory consumption.

JESS is available at no cost for academic use. A license was acquired from Craig Smith at Sandia National Laboratories.

3.4.3 FuzzyJ Toolkit & Fuzzy Jess

The FuzzyJToolkit has been developed by National Research Council of Canada's Institute for Information Technology (Orchard, 2006). It provides fuzzy functionality through Java and is integrated with JESS. FuzzyJess is available for academic use at no cost.

3.4.4 IUCBRF

"The Indiana University Case-Based Reasoning Framework (IUCBRF) is a freely available open-source framework, written in Java, to facilitate the development of case-based reasoning (CBR) systems. IUCBRF provides code to handle many of the general, domain-independent aspects of CBR systems, freeing developers to consider only the domain-dependent aspects of the system. The framework is designed to facilitate fast and modular development of CBR systems, providing a foundation for code sharing by those developing CBR systems" (Bogaerts, 2005).

The IUCBRF was chosen over several alternatives because: it was written in java (like the rest of the software developed in our project), it was the most mature java CBR software available, and it is available freely upon request under an open source license.

4. Analysis of the Problem

We began by looking at the major constraints on our project. We had to design and implement an intelligent soft-computing agent. It had to be capable of interfacing with another robot via the DPP. Then it had to recognize and treat the hypothesized defect via this interface. There were no requirements on what language the project should be written in. A major part of our problem was that the DPP and inter-robot diagnosis were still concepts in their infancy. Due to this, there were not many details available to us. We could not expect interface specifications. Because there were no robots built at the time we did our project we couldn't expect much specific information about possible failures or even a definite model of robot functionality.

Figure 2 shows a high-level view of how the DPP would allow our robots to communicate with each other and how the DPP would be connected to the internals of each robot.

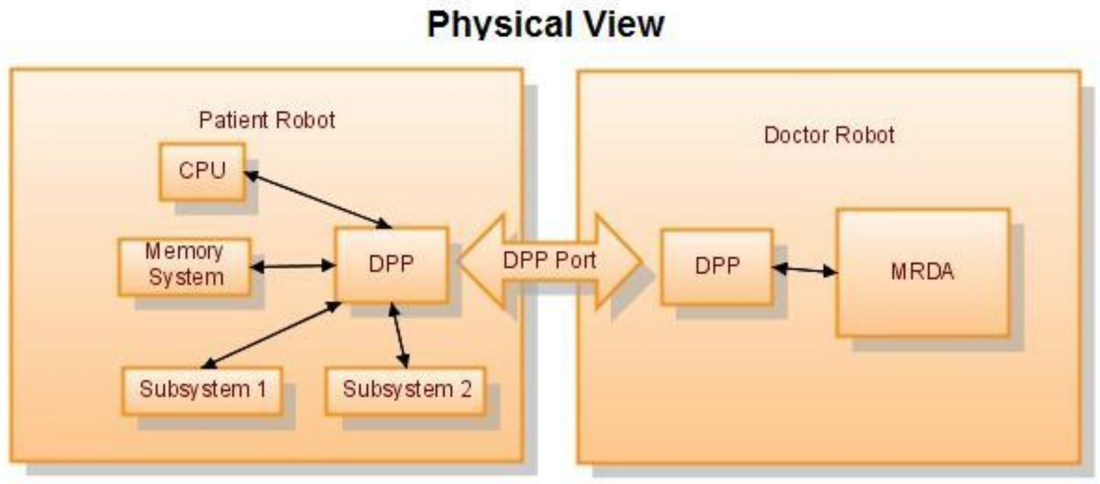


Figure 2 : High-level physical view of robot interaction via DPP

Since our project will be used as a proof-of-concept one of our focuses must be on making a system that is easily demonstrated. Our GUI must be intuitive and show all fundamental parts of the system operating. In many cases we needed to design our own models to work with, as none were currently available. Our decisions had to be guided not only by good sense, but also by standards in related fields. In selecting the types of tools we used to build the individual AI reasoners we were only concerned that they could be made to work. In an actual implementation more thought would go into selection. In our work we don't need to be too discriminating as we aren't worried about how well a single reasoner can diagnose or treat a problem, but how well the system as a whole (specifically the meta-reasoner) works.

We needed to create our own robot models. Working from some possible defect/mitigations we created further possibilities. We also needed to create a specification for how communication would be handled over the DPP.

4.1 Requirements

We created several categories of requirements based on discussion during our teleconferences with NASA. The most important requirements that had to be in the project were termed "Primary". Those requirements that would be "nice" to have but weren't necessary were "Secondary". Finally the requirements that we came up with that would add depth to the project but weren't directly related to the more important requirements were "Optional". These three categories of requirements were defined for both the Meta-Reasoner and the MRDA system as a whole.

4.1.1 Primary

The MRDA:

1. uses soft computing techniques;
2. employs an agent architecture;
3. contains at least one patient model;
4. has reasoners that operate on that model;
5. can retrieve information (symptoms) from the patient;
6. uses a meta-reasoner to select among reasoners;
7. contains at least one reasoner to create mitigations;
8. performs basic mitigation planning (decomposing the mitigation into actions);
9. sends the proper mitigation steps to the patient;
10. can be easily demonstrated using a GUI that shows the essential functions;
11. has a simple method for determining the success of a mitigation;
12. can store successful mitigations for later use;

The Meta-Reasoner:

1. performs pre-diagnosis reasoning about data;
2. can determine when to trigger reflex actions;
3. determines the set of reasoners to choose from;
4. makes an appropriate decision about which reasoner to use;

4.1.2 Secondary

The MRDA:

1. demonstrates the ability to swap models of patient robots (although these models may not be functional);

2. has a more advanced GUI that shows information about the progression of reasoning, states of robots, and the data being evaluated;
3. has a more advanced method for determining the success of a mitigation;
4. may attach information about successes to reasoners or specific mitigations;

4.1.3 Optional

The MRDA:

1. contains multiple patient models;
2. has multiple reasoners that act on each of these models;
3. performs more advanced mitigation creation, or has multiple reasoners for this purpose;
4. uses a more advanced reasoner to perform mitigation planning (decomposing into actions);
5. has additional GUI enhancements (for example, a "demo" mode that simulates various problems and mitigation cycles with minimal input from user);

4.2 Defect Scenarios

A defect scenario is all the information surrounding a defect experienced by a robot, most importantly it includes the symptoms exhibited by the robot with that defect. The symptoms are generally listed as "faults", which are signals the patient robot uses to indicate that something is wrong. We used defect scenarios to help design all the knowledge that would be required by the doctor robot to perform diagnosis and allow us to get feedback about that knowledge from our advisors at NASA. The defect scenarios provided the foundation for training all of our individual reasoners. Our advisors at NASA provided us with some sample scenarios where problems had occurred in the past on rover missions and also problems that were likely to occur on such missions. However, for the most part our scenarios had to be “invented” based on these provided scenarios, similar problem analysis (Spirit and Opportunity Mars rovers), and feedback from NASA on our own artificial (“made-up”) scenarios.

4.2.1 Provided scenarios

Diagnosis for some scenarios relies on a process called trending where the recorded data is analyzed. When signals exceed normal operating ranges “yellow-limits” and “red-limits” (according to the level of deviance from normal) the signals are recorded along with the time they occurred. Cross-referencing the yellow and red limits with other data that occurred at those times can help determine what may have cause the problem (or at least what else was going wrong at the same time). Many of our scenarios assume some kind of analysis like this has occurred and we use the notion of limits in defining whether the certain patient sensor values are good or bad (i.e., they indicate a fault).

4.2.2 Artificial scenarios

We decided to split all defect scenarios into subcategories that represent systems on the robot being diagnosed. This makes it easier for the Meta-Reasoner to perform a rough classification of the defect and will possibly help in deciding which specialized reasoner to select. The main subcategories are: Power, Mechanical, Communications, and Data Handling. We added to this list the categories of Reflex and Utility; the first is for defects that require reflex action and the second is for defects that are unique to the *type* of robot we are diagnosing. These scenarios were placed into an excel sheet during implementation for the purpose of training the reasoners (See Appendix A). The defect scenarios that our system handles are described below.

Reflex

Name: Return to Base
Description: The patient has already determined that it needs to return to base. For some reason, the patient is unable to return to base.
Faults: Return to Base Flagged
Treatment: Tow the patient back to base

Name: Drained Battery
Description: The battery has been drained of its power. The battery has become drained as a result of circumstance rather than a defect in the system.
Faults: Low Power Fault, Low Battery Charge
Treatment: Recharge the battery

Power Subsystem

Name: Dead Battery (Day)
Description: The lifespan of the battery has reached its limit. It is completely discharged, and incapable of further recharging. A low power fault will occur as the battery gradually fades. A High CPU Temperature Fault would occur due to the lack of power used for cooling.
Faults: Low Power Fault, High CPU Temperature Fault

Name: Dead Battery (Night)
Description: The lifespan of the battery has reached its limit. It is completely discharged, and incapable of further recharging. A low power fault will occur as the battery gradually fades. An electronics module low temperature fault would occur due to the lack of power for maintaining heat.
Faults: Low Power Fault, Electronics Module Low Temperature Fault

Name: Lunar Dust Covered Solar Panel
Description: The Solar Panel has been covered in lunar dust.
Faults: Inefficient Solar Panel Fault, Low Power Fault, Low Voltage Fault, Low Angle to Power Output Ratio

Name: Poor Solar Positioning
Description: The patient is poorly positioned to receive sufficient solar power to maintain its functionality. It should relocate to a better position or return to base.
Faults: Low Power Fault, Angle to Power Output Ratio High, Low Voltage Fault

Communications

Name: Directional Antenna is Out of Line of Sight
Description: The directional antenna is out of line of sight with the base.
Faults: Directional Antenna No Response Fault

Name: Directional Antenna is Broken
Description: The directional antenna is broken.
Faults: Directional Antenna No Response Fault, Directional Antenna Verification Fault

Name: Omni-Directional Antenna is Out of Range
Description: The omni-directional antenna is out of range from the base.
Faults: Omni-Directional Antenna No Response Fault

Name: Omni-Directional Antenna is Broken
Description: The omni-directional antenna is out range with the base.
Faults: Omni-Directional Antenna No Response Fault, Omni-Directional Antenna Verification Fault

Processor

Name: Processor CPU Damaged Due to Overheating
Description: The CPU overheats and becomes damaged.
Faults: CPU High Temperature Fault, Electronics Module Temperature Over Limit

Name: Firmware Corruption Due To Solar Radiation
Description: The radiation protection coating on the firmware memory block has faded or been made ineffective. It was corrupted in the recent solar radiation storm. This results in memory corruption of the firmware. Thus the system is idle processing more than it should be and increasing the temperature of the module.
Faults: Boot Failure Fault, CPU High Temperature Fault, Increased Idle Processing State Fault, Electronics Module Temperature High

Transportation

Name: Broken Wheel
Description: A wheel has broken.
Faults: Wheel Encode Fault, Anticipated Movement Fault

Name: Broken Drive Shaft
Description: The drive shaft is broken.
Faults: Wheel Encoder Fault, Drive Shaft Encoder Fault, Anticipated Movement Fault, Wheel Motor High Temperature Fault

Name: Broken Motor
Description: The motor driving the wheel has been broken.
Faults: Wheel Encoder Fault, Drive Shaft Encoder Fault, Motor Encoder Fault, Anticipated Movement Fault

Name: Positioning System Malfunction
Description: The system used to determine the position of the robot is returning the same position regardless of its position.

Faults: Anticipated Movement Fault
Supplementary Data: No other systems are returning faults

Utility

Name: Broken Camera
Description: The specialized camera is defective and broken.
Faults: Camera Verification Fault, Camera No Response Fault

Name: Broken Arm
Description: A specialized robot arm is broken.
Faults: Arm Verification Fault

Name: Broken Detector
Description: A specialized spectral imaging detector has been broken.
Faults: Detector Corrupted Data Fault

5. Design and Implementation

5.1 MRDA Design

Our application can be divided into three layers: presentation, application, and data (see Fig. 3). The presentation layer consists of our graphical user interface. The graphical user interface (GUI) allows the user to run the simulation. Since there was no existing simulation in which to test our meta-reasoning diagnosis agent, we developed a framework and simulation for the robots to interact. The simulation included simulated robots and a communication network. The application layer is where the simulation of the robots and diagnosis takes place.

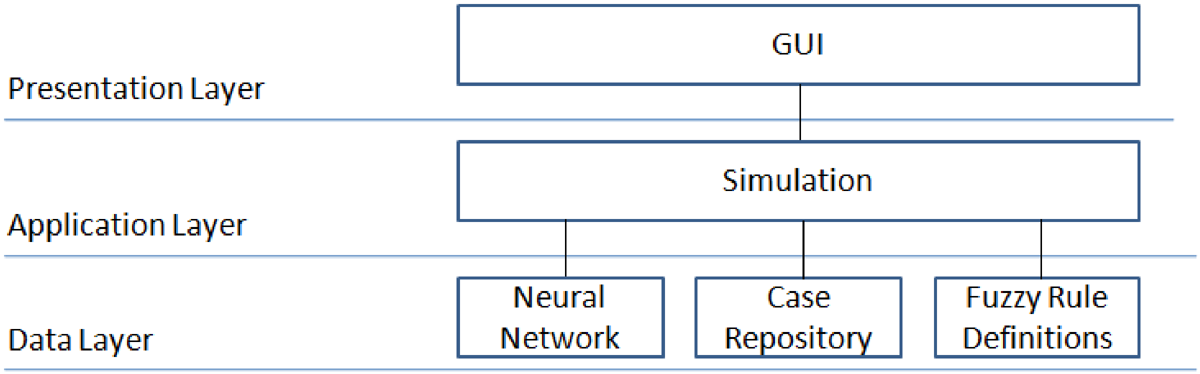


Figure 3 : High-level representation of MRDA

The GUI interacts with the simulation through an interface, which allows the user to change the symptoms of the broken robot, specify an operating condition, and initiate diagnosis. The data layer for our prototype is implemented using local storage on the system running the software, where the neural network, case repository, fuzzy rule definitions, and other basic data is stored. The simulation loads and uses this data when it is functioning.

5.1.1 Simulation Architecture

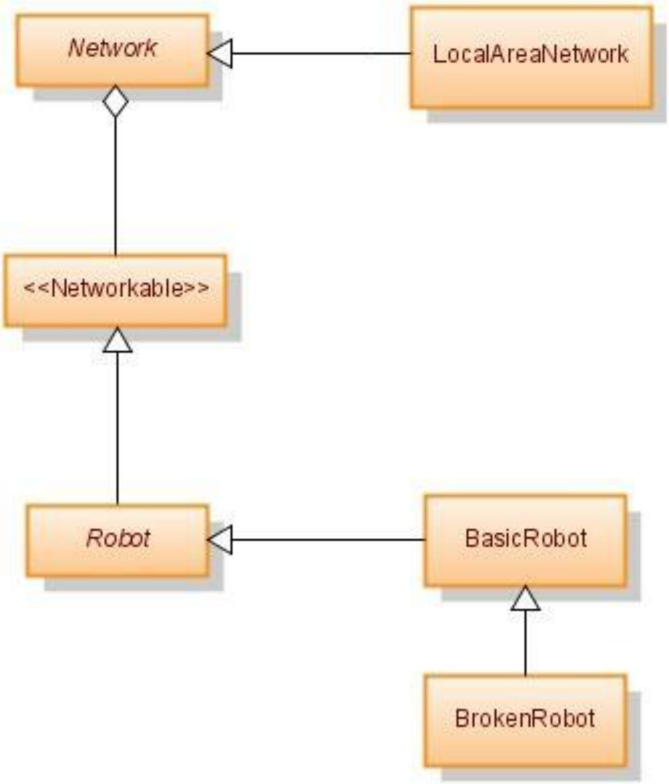


Figure 4 : High-level simulation class diagram

Figure 4 shows a class diagram at the highest level of the simulation. The simulation consists of a network and two robots, a BasicRobot and a BrokenRobot, the doctor and patient robot, respectively. Both robots are registered with the simulated network when the application starts. As members of the network, they receive any message broadcast by another robot on the network. By having the Robot class implement the Networkable interface, we ensure that all robots can join a Network, and thus communicate with other robots through that network. This communication is necessary for initiating diagnosis procedures, transferring data, and issuing commands to other robots.

5.1.2 Robot Communication and Messages

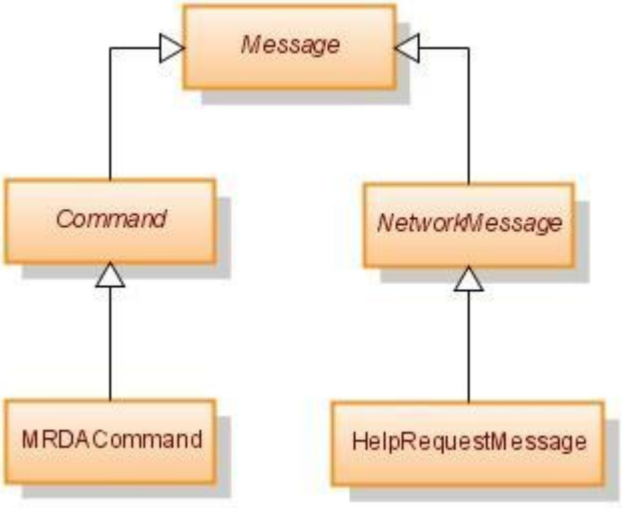


Figure 5 : Inheritance hierarchy of messages

Communication is performed by sending messages across the network. Figure 5 shows the inheritance hierarchy of messages. We define two major types of messages: commands, and network messages. Commands are sent to tell another robot to perform some task. Network messages are used for broadcasting general messages, such as a help request message. The broadcast of a help request message is the start of the simulation. The patient robot will broadcast a help request message to all robots on the network. In the case of our simulation, we have only the doctor robot and the patient robot. When the doctor robot receives this message, it retrieves information about the sender included in the message. With that information, it establishes a connection with the patient robot and begins its diagnosis procedures.

Similarly, when a command is broadcast on the network, all robots except the broadcasting robot receives the command and perform it. For our prototype, the robot does not actually perform any functions as a result of a command. It simply processes the command by outputting to the log that the command was received and performed.

5.1.3 Functional Flow

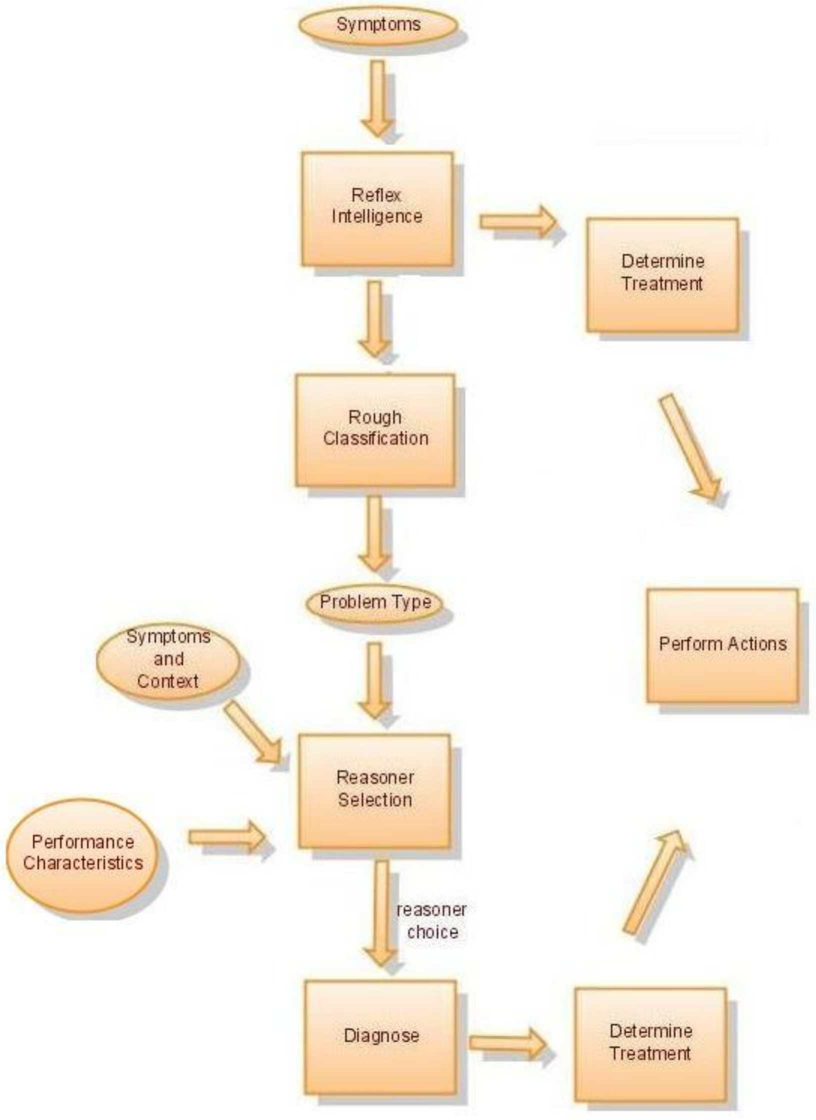


Figure 6 : Flow of MRDA functions

Once the doctor robot has established a connection with the patient robot, it begins its diagnosis procedures. Figure 6 is a functional diagram of the diagnosis process. The doctor robot requests a set of symptoms from the patient robot and immediately checks to see if it has a reflex reaction. Symptoms are passed as a list of flags that have been activated by the patient robot. These flags may be activated by various conditions such as temperatures rising pass a certain threshold. If a reflex reaction is found, it immediately determines the appropriate reflex treatment and performs it. If the reflex diagnoser cannot handle it, a rough classification is performed to determine the subsystem that the problem is most likely to be in. Using the rough classification and performance characteristics described in section 5.2.1, one of the three diagnosers is chosen for diagnosis. Once the diagnosis is complete, a treatment is found for the problem. The treatment is executed by broadcasting each command in the treatment.

5.1.4 Data Flow

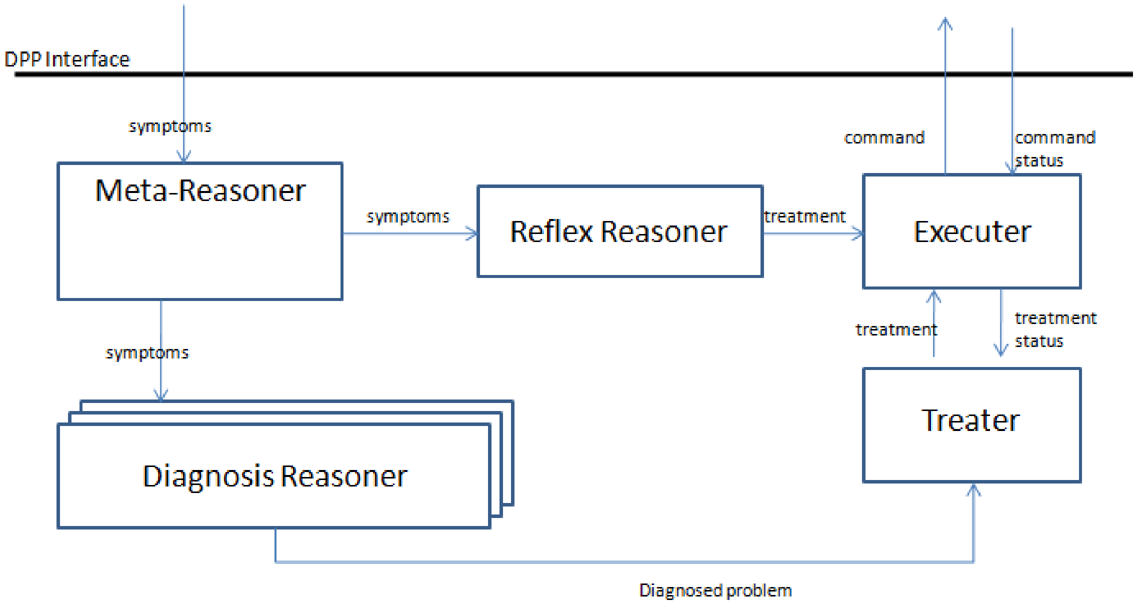


Figure 7 : Flow of data in the MRDA

From a data flow perspective, as seen in Figure 7, the functional diagram corresponds to a few major components in our diagnosis agent. When the symptoms have been retrieved from the patient robot, they are passed to the Meta-Reasoner. The Meta-Reasoner first queries the Reflex Reasoner to see if it can handle the diagnosis. If it can, control is passed to the Reflex Reasoner, and the reflex treatment is given directly to the Executer for execution. If the Reflex Diagnoser is not able to handle the problem, the

Meta-Reasoner performs its meta-reasoning, described later in this document, and passes control to the appropriate diagnosis reasoner. The diagnoser will then produce a problem which it passes to the Treater to determine the appropriate treatment. The Treater then passes the appropriate treatment to the Executer for execution. The Executer takes a treatment, and sends each command in the treatment to the appropriate robot.

5.1.5 Additional Components

The Treater in our application performs its function in a simple manner since it is not the focus of our project. It performs a direct mapping for each possible problem to its treatment. It determines this treatment, and passes it to the executer. A treatment consists of an ordered set of command messages that specify who the command should be executed by.

The Executer is also trivially simple for the same reason. It performs its function simply by executing the provided treatment's commands by delivering, in order, the command message to either the patient robot or the doctor robot. When it issues a command, it sends it directly to the patient robot. If the command specifies that it should be performed by the doctor, the executer sends it to the doctor's message processing system. For example, some treatments command the doctor robot to secure the patient robot for towing.

5.2 Diagnosers

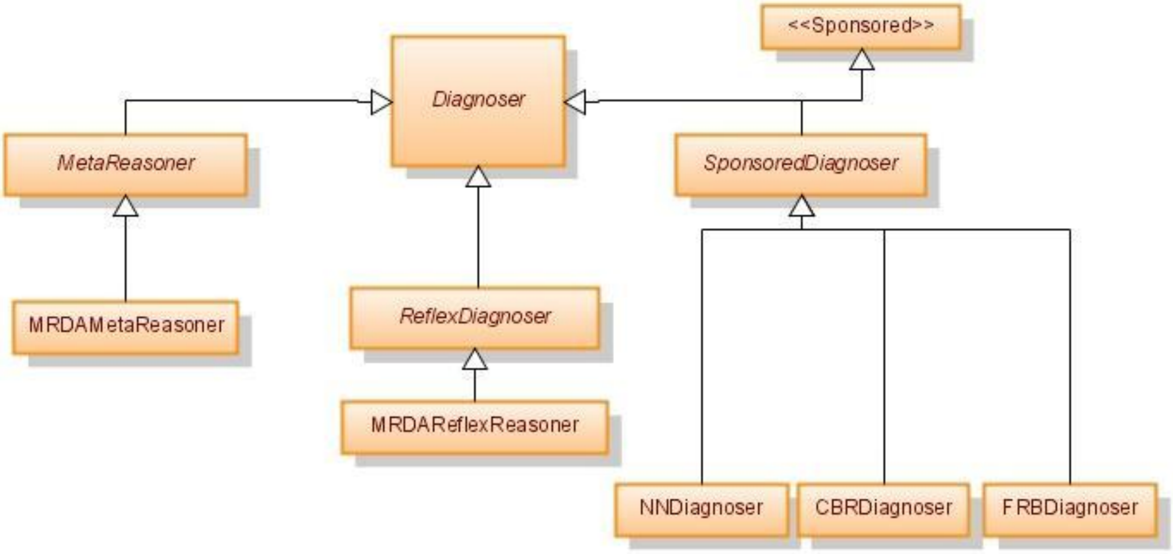


Figure 8 : Inheritance hierarchy of diagnosers

Figure 8 shows a class diagram of the inheritance hierarchy of the diagnosers. All diagnosers inherit from the Diagnoser abstract class, which ensures that all diagnosers have the diagnose() method which accepts a collection of Symptoms as input. There are three major types of Diagnosers: meta reasoners, reflex-diagnosers, and sponsored diagnosers. They are all diagnosers because they all take a set of symptoms and perform diagnosis. The Meta-Reasoner chooses to perform its diagnosis by delegation. The reflex diagnoser chooses to produce a treatment and immediately pass control to an Executer, rather than passing a diagnosed problem to a Treater. Finally, the sponsored diagnoser implements the Sponsored interface described below. This allows it to be queried for a vote of confidence representing the diagnoser's confidence that it can diagnose the problem. We implemented three sponsored diagnosers: a neural network, a case-based reasoner, and a fuzzy-rule-based reasoner.

5.2.1 Meta Reasoner

The Meta-Reasoner is the primary diagnoser in our agent. It starts by checking to see if the reflex diagnoser can handle the problem. If not, then it performs a rough classification which determines the subsystem that the problem is likely to be in (e.g., Power Subsystem). Having a rough classification allows the meta-reasoner to choose a reasoner based upon the reasoner’s coverage of the subsystem. Then, it queries the sponsored diagnosers for their votes of confidence. Given a set of desired performance characteristic and subsystem classification, the sponsored diagnoser produces a vote. The highest vote is assumed to represent the best choice for a diagnoser. The meta-reasoner then delegates the diagnosis to the chosen diagnoser.

Rough Classification

Rough classification is performed by a hard-coded check for the presence of a specific symptom that indicates a problem in a specific subsystem. By analyzing the symptoms associated with problems in each subsystem, we identified similarities. For example, problems in the power subsystem should all exhibit the Low Power Fault symptom. During rough classification, we check to see if the Low Power Fault was present. If it was true, then we would roughly classify the problem to be in the power subsystem. In some cases, it was more complex. In the case of the Command and Data Handling subsystem, we looked for the presence of a High CPU Temperature Fault, and the absence of an Electronics Module Low Temperature Fault. In another case, the presence of any of two symptoms was sufficient for the rough classification.

Of course, a more robust implementation of rough classification could be used in a future version, but given our limited knowledge of the domain, we decided that a hard-coded boolean classification would be appropriate.

Sponsor Selection Mechanism

The Sponsor Selection mechanism, as described in the paper “A Knowledge-Based Selection Mechanism for Strategic Control with Application in Design, Assembly, and Planning” by Punch et. al. (1995), uses the rough classification as well as information about the desired performance characteristics to determine which diagnosis reasoner is the best choice. Each of the sponsored diagnosers provided sponsor functionality, meaning that it could be queried to return a vote of confidence on its ability to diagnose the problem. The meta-reasoner queries each sponsored diagnoser with the rough classification and performance characteristics. We have two possible performance characteristics: accuracy and speed. In future versions, this could be extended to describe other features about the diagnoser’s performance.

We implemented the voting mechanism such that subsystem coverage would dominate the performance characteristics. However, we allowed the performance characteristics to be the deciding factor between two diagnosers that both covered the given subsystem. If a diagnoser covered the given subsystem, it would add 1.0 to its vote. If it met any of the performance characteristics, it would add 0.1 to the vote. The meta-reasoner compares the votes, and passes the diagnosis to the diagnoser with the highest vote. In the case of ties, the decision defaulted to the following order of priority: case-based diagnoser, neural network diagnoser, and fuzzy rule based diagnoser. The default ordering is based upon reliability and coverage of the reasoners. The reasoner that has the most coverage is given more priority in a tie because it has the highest probability of covering the relevant subsystem.

5.2.2 Neural Network Diagnoser

The Neural Network is implemented using the Java Object-Oriented Neural Engine (JOONE). The network is implemented as a three layer neural network. The first layer is used for input and is of the size of the number of symptoms. Each input represents a different symptom. The input layer is connected using a full synapse with the hidden layer. A full synapse, as shown in Figure 9, consists of a synapse for every combination of two nodes consisting of a node from each layer. A full synapse allows the training of the network to determine the relevance of a relationship between nodes. The irrelevant synapses will be trained to have lower weights than the more relevant synapses. The hidden layer is connected to the output layer using a full synapse.

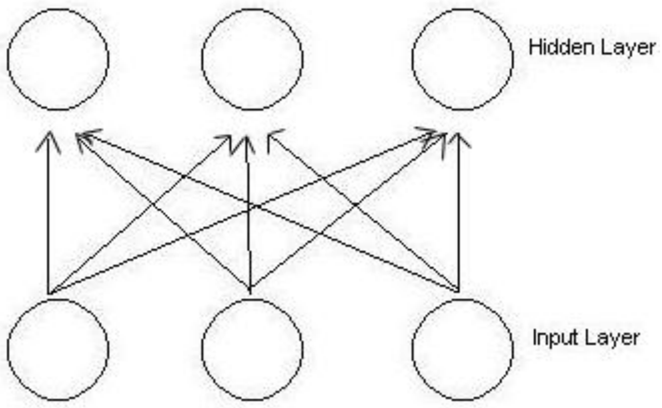


Figure 9 : A Full Synapse between two layers of three nodes

A training application was created to train the network. The training application trains the neural network by adding an input synapse to the input layer, and a teaching synapse to the output layer. Then it generates a set of training data, and converts it into the appropriate format for training, double arrays. Training data is generated by randomly producing symptom values within the appropriate interval for a given defect. For example, if a defect exhibits high CPU temperatures, a random temperature is generated within the high temperature interval. Each training scenario has an index by which it identifies what problem it is. The neural network is trained to maximize the output node with that index for that input. Training is performed with a few sets of slightly randomized training data with several thousand training cycles for each set in order to ensure that the neural network is trained to handle ranges of data for symptoms such as temperature readings.

Once trained, a neural network can be saved for use in the application. The neural network is loaded from a file, and given an input and output synapse. Given a set of symptoms, a double array is created as input to the neural network. The neural network is run, and the index of the output node with the highest value corresponds to the problem that the neural network has diagnosed. The index value is converted into an internal problem object, which the neural network diagnoser passes to the treater for treatment.

In order to design the neural network's architecture, and train it, trial and error was used. It was difficult to find resources regarding the best design or training method for a neural network. Through experimentation, a hidden layer of approximately 50% more nodes than the input layer was found to be necessary for reasonable output. Training was an even more challenging problem, with only two real parameters to work with: learning rate and momentum. According to the JOONE manual, the standard procedure for this process is to run several neural networks in parallel with varying parameters to determine the optimal result. We essentially did this, but without the automation described in the manual (Marrone, 2007). The randomized nature of the training set added another level of complexity to the problem. Ultimately, we settled for a neural network that successfully diagnosed six of the seven problems it covered. Since after a month of training, a perfect neural network was not successfully trained. The seventh is covered by the Case-Based Reasoner, which demonstrates the need for multiple reasoning techniques, and the importance of meta-reasoning.

5.2.3 Fuzzy Rule Based Diagnoser

The Fuzzy Rule Based Diagnoser (FRB) is implemented using FuzzyJ and Jess. We use a Fuzzy Rete engine provided by FuzzyJ, and used the lisp-like language to define the rules, fuzzy variables, templates, and queries. At the start of the application, the fuzzy engine is loaded, the rules are parsed and then executed in the engine. When the FRB is given a set of symptoms, it converts those symptoms into facts which are asserted into the working memory of the engine. The engine then executes the rules . Rules are executed using the default Mandani Min Max Min algorithm. A query, as defined in the parsed file, is then used to retrieve the resulting diagnosis. The diagnosis is then converted into a problem which is passed on to a treater for treatment.

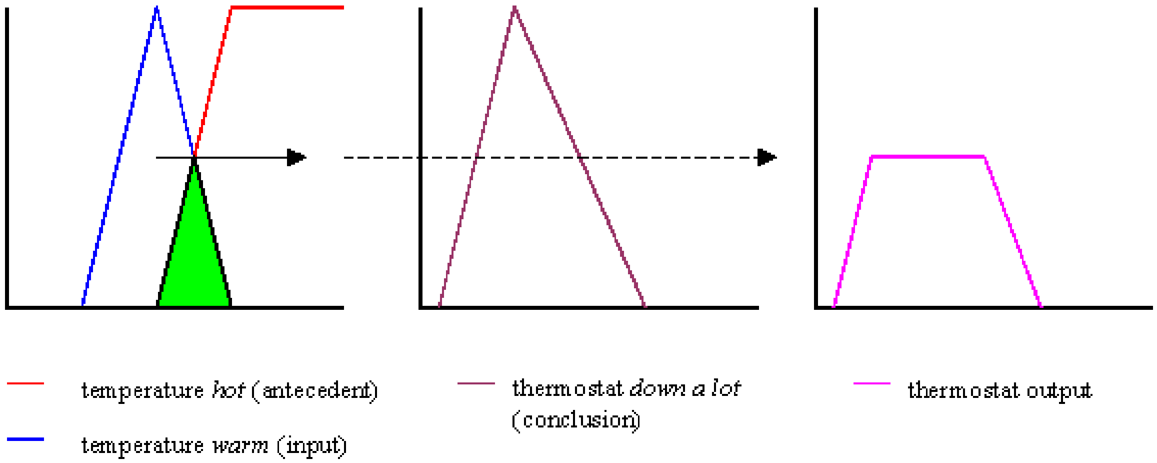

temperature *hot* (antecedent)
temperature *warm* (input)
thermostat *down a lot* (conclusion)
thermostat output

Figure 10 : Mandani Min Max Min Rule Execution (Orchard 2006)

Rules are evaluated using the default Mandani Min Max Min method. Figure 10 depicts how the rule: “**If** *temperature is hot* **then** *turn thermostat down a lot*.” is used with the input *temperature is warm.* The consequence *‘turn thermostat down a lot’* is scaled down due to the ‘*warm*’, not ‘*hot’*, input. (Orchard 2006)

As you can see, the fuzzy set for temperature warm is compared with the fuzzy set for temperature hot, and the resulting green triangle’s height is used to clip the fuzzy value for turning the thermostat down a lot.

We use fuzzy-rules to evaluate temperatures, battery charge, and the ratio between the angle from the normal of the solar panel to the sun and the power output. By defining a fuzzy set for, hot, cold, and normal temperatures, good and bad ratios, and low, medium, and high battery charges, we can fuzzily handle the input. For example, one of the rules involves a bad ratio. We then provide a fuzzy set representing the range of ratio we have as input, and compare it with the fuzzy set for bad ratio to get a boolean membership as a result. Due to the lack of sufficient expert knowledge, our fuzzy rules are simplified such that the conclusion of the fuzzy rules are simply discrete binary results, rather than a fuzzy conclusion. With the appropriate knowledge, complex fuzzy rules could be added to the system.

5.2.4 Case-Based Reasoning Diagnoser

The Case-Based Reasoner (CBR) was built using the IUCBRF (Bogaerts, 2005) Java-based framework. Initially a basic CBR was built based on examples provided with the IUCBRF package to get an early estimate of the feasibility of working with the package. Most advanced functions of the basic CBR, such as performance monitoring, adaptation, and case-base maintenance, were disabled because they were unnecessary for our application. This basic implementation was set up to use a flat case-base which simply stores the cases in a file (as opposed to more structured methods of storing cases such as a binary tree).

The next step was to add a way of inputting cases and problems to be solved into the CBR. Most example implementations of IUCBRF systems used a “generator function” that produces random cases and problems. This was unsuitable for our needs and eventually a way of entering case and problem information through a graphical interface was employed. Once we could get this data into the system we needed a way of storing it for later use. Although the CBR was storing cases in a flat-case-base, the storage would be rebuilt each time the program started and data would have to be entered again. A built-in method of saving and loading the case-base was found in the IUCBRF package and implemented. Unfortunately the saving process produced a file which contained information about the system it was built in and could not be moved to another CBR. This meant that the case-base file could not be built in the basic CBR and then later moved into the CBR integrated into the MRDA.

The basic CBR code was migrated into the MRDA project and the case-base rebuilt by hand via the graphical interface. Because of how tedious the process of entering cases was it would be desirable to have a simpler method of inputting them, such as hard coding the cases and having the CBR load them in. However, because of the small number of cases that had to be added it was determined that little time, if any, would be saved by the time an easier method of adding cases was implemented.

At this stage, basic testing of the CBR was started to determine its accuracy with sample data. Some symptoms are Boolean values and others could take a value in a numeric range. During testing it was noted that the CBR would often confuse problems which had similar inputs where a numeric symptom had a slightly different value.

We came up with three solutions to this problem: additional training cases could be created which would “train” the CBR to recognize the boundaries of numeric symptoms that indicated a meaningful change, the weights (degree of importance to the final result) of the numeric symptoms could be tweaked so that they wouldn’t affect the result as much, or the numeric symptoms could be replaced in the CBR with Boolean symptoms.

The first solution would require a significant number of additional cases to be created and entered into the system and might suffer the same problem of favoring numeric similarities too much over Boolean similarities. The second solution risked weakening the influence of the numeric symptoms to the point where meaningful changes might not affect the result and it also would require much trial-and-error testing to find acceptable weights. The third solution was deemed best as there were already various boundaries of the numeric values which indicated important changes and that would be easy to represent as a set of Boolean symptoms (these numeric symptom boundaries were already available to the reasoners). The numeric symptoms were each replaced with four Boolean symptoms indicating whether or not the value was between certain boundaries and the case-base was rebuilt.

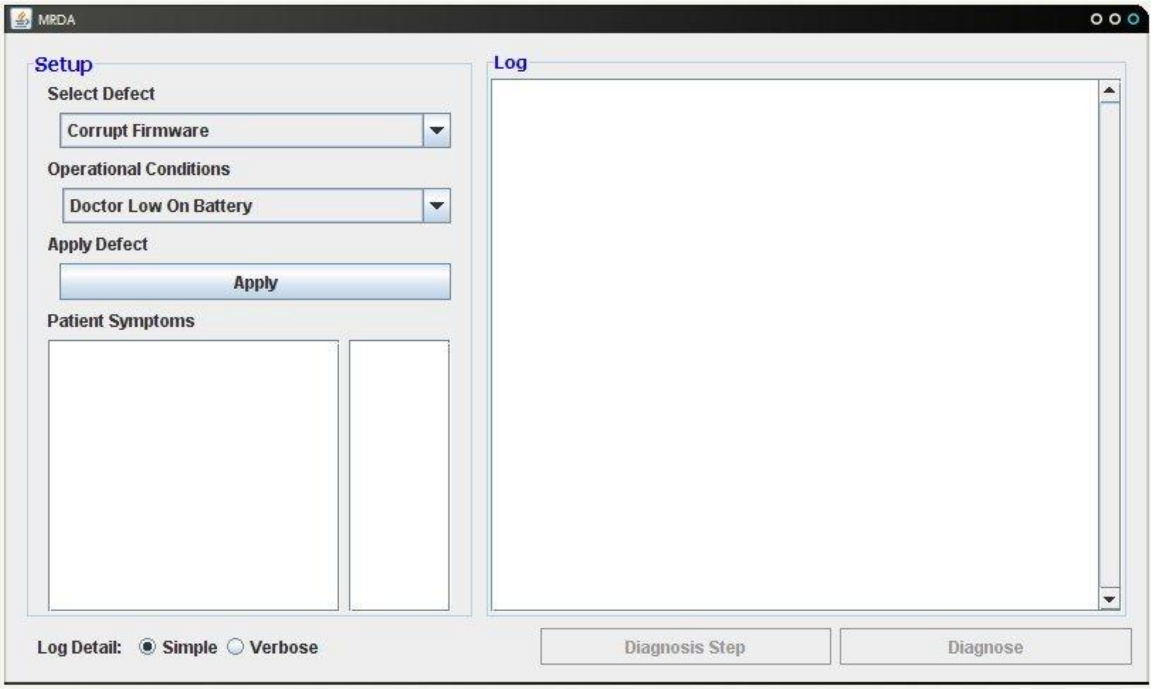
Once the CBR was producing correct results on sample data it had to be fully integrated into the MRDA code. The CBR had to be coded to return its diagnosed problem to a treater in a specially defined “Problem” type. Also the CBR was required to take input from the MetaReasoner as a list of symptoms. First the CBR had to be modified to accept a problem from the code rather than through a graphical interface (as was used during testing). The list of input symptoms provided by the MetaReasoner was not the same as the symptoms expected by the CBR (because of the numeric to Boolean symptom modification). Therefore the next step was to create a mapping between the symptoms in the list provided by the MetaReasoner and their expected positions for the CBR. The additional Boolean symptoms were then added based on the provided numeric symptom values. Once this was done the CBR was fully integrated and functional.

5.2.5 Shared Knowledge Format

In order to keep our framework robust, and reduce the time spent adding support for each of the diagnosers, we defined a common format for representing information about symptoms, problems, and scenarios. Since each diagnoser used a different reasoning technique and used a different library, they each required different formats for the same information. By providing a common format, we could define all the symptoms, scenarios, and problems just once and use it throughout the framework. In order to bring that data into a format usable by the specific reasoner, a conversion was necessary. Each diagnoser converted the data into a format it could use, and generated a diagnosis. Then, it converted that resulting diagnosis into the shared knowledge format. This allowed the treater to handle answers from any of the three diagnosers without any complications.

5.3 GUI Design

The Graphical User Interface went through many revisions during its design and implementation. The first few iterations were produced on paper and discussed during our weekly meetings. We focused on usability and whether the design would meet the requirements for the interface. The next stage was to produce a graphical but nonfunctional prototype to judge how a final implementation would look. These designs were also presented to our advisors at NASA for their feedback. Over the course of design the GUI became more feature-rich, however during implementation some features of the design had to be removed in order to simplify interfacing with the system code and because of time constraints. Once a final design was settled on a GUI “walkthrough” was presented during a NASA teleconference to get feedback and approval for the interface and how interactions with the software happened in general.



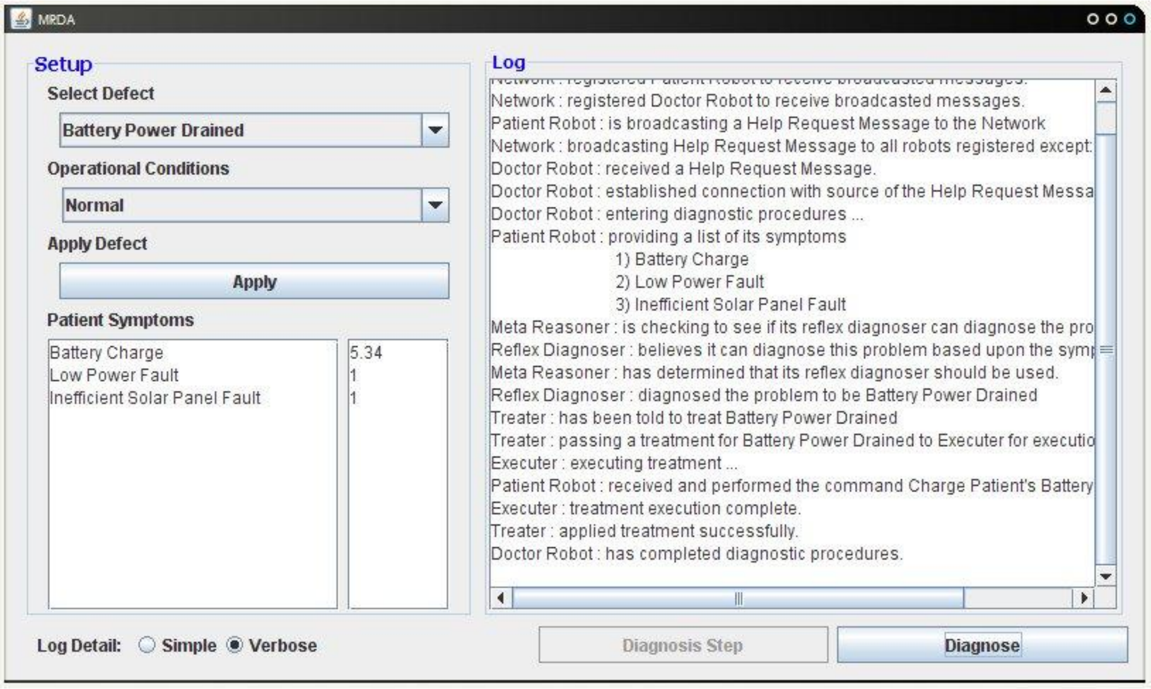
Select Defect: Corrupt Firmware

Operational Conditions: Doctor Low On Battery

Log Detail: ● Simple ○ Verbose

Figure 11 : Screenshot of Graphical User Interface

Figure 11 shows an image of the screen presented to the user when the program is launched. The interface has selection boxes on the left hand pane for choosing testing scenarios. The scenarios include a defect (which sets the symptoms present on the patient robot) and an operational condition (which modifies the criteria by which reasoners are selected). Once a scenario is chosen, selecting the “Apply” button initializes the system and applies both the defect and the conditions the user picked. As soon as this happens the window below, labeled "Patient symptoms", lists the various values present on the patient that are considered to be symptoms of the chosen defect. Also by selecting the "Apply" button the "Diagnose" button in the lower-right of the GUI is enabled allowing it to be selected. Selecting this button will display a step-by-step list of the actions taken by the robots including communication, diagnosis, and treatment.



Select Defect: Battery Power Drained

Operational Conditions: Normal

Patient Symptoms

Battery Charge	5.34
Low Power Fault	1
Inefficient Solar Panel Fault	1

Log Detail: ○ Simple ● Verbose

```
Network : registered Patient Robot to receive broadcasted messages.
Network : registered Doctor Robot to receive broadcasted messages.
Patient Robot : is broadcasting a Help Request Message to the Network
Network : broadcasting Help Request Message to all robots registered except:
Doctor Robot : received a Help Request Message.
Doctor Robot : established connection with source of the Help Request Messa
Doctor Robot : entering diagnostic procedures ...
Patient Robot : providing a list of its symptoms
                    1) Battery Charge
                    2) Low Power Fault
                    3) Inefficient Solar Panel Fault
Meta Reasoner : is checking to see if its reflex diagnoser can diagnose the pro
Reflex Diagnoser : believes it can diagnose this problem based upon the symp
Meta Reasoner : has determined that its reflex diagnoser should be used.
Reflex Diagnoser : diagnosed the problem to be Battery Power Drained
Treater : has been told to treat Battery Power Drained
Treater : passing a treatment for Battery Power Drained to Executer for executio
Executer : executing treatment ...
Patient Robot : received and performed the command Charge Patient's Battery
Executer : treatment execution complete.
Treater : applied treatment successfully.
Doctor Robot : has completed diagnostic procedures.
```

Figure 12 : Screenshot of the GUI showing an output trace

The detail of this output can be increased by switching from "Simple" to "Verbose" log detail in the lower-left. Figure 12 shows the GUI displaying output in verbose mode. Selecting "Diagnose" again refreshes the log window with the updated log detail mode. By switching from "Simple" to "Verbose" mode the "Diagnose Step" button is also enabled. Selecting this button produces one line of output to the log at a time, providing a slower paced interaction with the system which is ideal for presentation.

5.3.1 GUI usability evaluation

Although alternative GUI designs were presented to our co-advisors at NASA for feedback, we also wanted an idea of how intuitive and usable the final GUI was for a user with very minimal knowledge of the project. This would give us an idea of the usability of the physical design of the GUI and the presentation.

For the usability study a procedure was created to lead a test subject through our user interface. The goal was to note any specific difficulties or observations that the subjects made when we encouraged exploration of the interface. Background knowledge of the project was limited to prevent the function of the GUI being derived from project details rather than the physical layout of the GUI. The interviewer procedure is as follows:

1. Give general description of domain:

   > "Our project was to simulate the interactions between two of the robotic rovers used by NASA on missions to the moon. One of the rovers is the 'doctor' and one is the 'patient'. The patient has a problem and the doctor has to find out what it is and fix it."

2. Open the GUI.

3. Explain that this window is the user interface for our project.

4. Ask subject what they think the user interface displays (what information it makes visible).

5. Ask subject what actions the user interface allows them to make (how can they interact with it).

6. Ask the subject if they think they could run a simulation of our project just based on seeing the user interface and not knowing all the details of it. If they subject does not see how to run a simulation yet, show them.

7. Ask the subject if the understood in general what happened when the simulation ran (defects applied during setup, lots of output happens when they click diagnose, etc.).

8. Ask the subject if they think they could change the way the simulation was

run or displayed. (we want to see if they notice that the defect drop-down changes the patient problem, the difference between simple and verbose log modes, and the option to diagnose step-by-step).

9. Go through a verbose output line-by-line asking if the subject has any idea what each could mean (obviously details that are too domain oriented won’t be intuitive, but we are looking to see that the log output makes the process clear).

For subjects, friends and roommates of the experimenters were used (four subjects in total). None of the subjects were very computer savvy, but they had used computers for general communication, word processing, games, etc. All that the subjects knew about our project was the information given to them in the procedure above. The subjects were led through the procedure above and their reactions were noted.

Results

For the most part the subjects were able to identify the information that was displayed on the screen and the function of all parts of the GUI without even experimenting with the interface. There were some exceptions to this however. All subjects were unaware of the meaning of ‘verbose’ and therefore did not understand its effect on the log until they experimented with it. Also one subject was not sure whether the symptom list was the symptoms the patient exhibited or the symptoms the doctor diagnosed. The operational conditions selection box was at first not understood by any subjects, however most were able to determine its meaning (that it was a condition that affected the doctor) upon inspecting its contents. All subjects thought that the log output was understandable. Despite not knowing any details of the project the subjects were able to describe what was happening by reading through the log output.

Probably the most important observations during the study was that all of the subjects originally thought that multiple defects could be applied (probably due to the symptom list being a large list box and thinking of the apply button as “ADD”). Also the subjects did not know what “diagnose step” would do until they had clicked it multiple times and observed its functionality. Overall the subject’s initial understanding of the system (before experimenting with it) was greatly increased by mentioning that the GUI was a simulation of the robot interactions. The “simulation” word being key.

In the case of this interface being used during a presentation, most of these issues would not be a problem due to increased knowledge of the project on part of the audience and the fact that the interaction is being led by someone knowledgeable.

However, these results still suggest that some minor changes could be made to the GUI to make it even more intuitive. The diagnose and diagnose step buttons can be moved into the left pane to eliminate confusion between their function and that of the apply button and to make them more visible. Also the left pane could be renamed ‘simulation’ and include a subpane for ‘running’ the simulation which would contain the diagnose buttons. The diagnose button could be renamed ‘full diagnose’ to highlight the difference between it and ‘diagnose step’ however, this could just add to the confusion in that people may think ‘diagnose step’ somehow results in incomplete diagnosis. Verbose log detail could be renamed to ‘detailed’. In order to give the impression that only one defect can be active at a time the symptom list could be renamed to ‘symptoms for defect []’ where the blank is the name of the symptom they apply. That way there is less chance users will think the list can display symptoms of multiple defects. However, these modifications should only be appropriate if the interface would be used by individual users not familiar with the work (which is not the case in the intended use during presentations).

6. Evaluation

The goal of our prototype was to explore the use of a Meta-Resoner to select between different approaches to reasoning. To that end, we discovered a variety of concerns and benefits that should be considered in an implementation of this technique.

One of the ideas we had was to separate levels of reasoning. There was the low level reasoning of reflexive intelligence and the higher level reasoning of the neural network, case-based reasoner, and fuzzy rule-based reasoner. This created a multiple-level reasoning strategy where urgent problems could be addressed without wasting time and resources on higher-level reasoning. We had difficulty in finding uses for the reflex reasoner. While we ultimately did find a couple of scenarios that could be reflexively handled, we discovered that most of the obvious reflex scenarios were handled by the patient-side diagnosis systems. Essentially, the lower levels of reasoning were already being performed on the patient robot. Nonetheless, we believe using different levels of reasoning is a good strategy; however, future implementations should have their main focus on combinations of higher levels of reasoning.

Basic testing was done through the GUI simulation interface. All possible selectable defect scenarios resulted in the correct output (the correct selection of reasoner, the correct diagnosis, and correct treatment selection).

We also discovered that the meta-reasoning strategy using sponsor selection was an effective technique. Using performance specifications and rough classification to choose between reasoners with different problem coverage proved effective. In many cases, we found that one reasoner would have gaps in its ability to successfully diagnose problems in a certain subsystem. However, there was always another reasoner that could correctly diagnose the defect. Using the Meta-Reasoner to select the appropriate reasoner given the scenario allowed the project as a whole to have a 100% success rate diagnosing defects, while an individual reasoner would not perform as well on its own. One issue with this is that in our project the rough classification had to be hard coded, requiring human analysis. In larger scale projects with more knowledge, it would be difficult to find simple rules by hand to do rough classification. Automating the process would likely require searching over a set of defects and symptoms to find commonalities (the smallest set of symptoms which are common to the largest group of defects). This is a challenge that should be considered before implementing future projects.

As the GUI was tested above in section 5.3.1: GUI usability evaluation, we will only summarize the significant result here. Overall the user interface was intuitive and understandable for subjects. In the case of a presentation of this work to a NASA audience the interface would be even more understandable because much more would be explained by the presenter. The biggest problems were that subject thought multiple defects could be applied at once, they did not know what ‘verbose’ log detail meant until they used it, and they did not know what the ‘diagnose step’ button would do until they used it.

7. Conclusion

We produced an application that successfully meets all of the primary requirements (See Section 4.1 Requirements) except for detecting a successful outcome and storing it for later use (Primary goal #12 for the MRDA). Although storing these successful actions was a primary requirement it became clear during the late design stages and implementation that our simulated environment (the patient robot) was not complex enough, and our treatment design process not unique enough, to merit implemented detection and storage of successful cases. Determining successful treatment is a simple matter of comparing the defect identified by the diagnosis to the initial defect applied to the patient. The treatment produced for the patient is a simple function of the diagnosed defect. Because there is no significant adaptation of either defect or treatment, storing successful cases would in effect just store multiple copies of the same cases already known to be successful.

Storing successful cases is a potential problem for any implementation of this system however because as time goes on more cases accumulate in the case-base and eventually all the knowledge represented in more advanced reasoners is transferred to the case-base rendering the advanced reasoners useless. For this goal to be successfully implemented diagnostic and treatment reasoners must be sufficiently advanced such that their knowledge could not otherwise be represented by anything but an exceedingly large set of cases (for example they could use a complex model of patient symptoms to predict behavior and uncover previously unspecified problems)

Our application provides a proof-of-concept for using a central MetaReasoner to decide amongst several reasoners. It also provides a robust and extensible framework for simulating robotic diagnosis and treatment. Possible extension of this project include a formal method for classifying (and maybe determining) the properties of different reasoners for the purpose of communicating these properties to the MetaReasoner and allowing it to make more informed decisions. Another excellent extension would be the construction of a richer simulation space for testing reasoning programs. This would include: complex realistic models of patient systems, representations of the external environment and its effects, models of interaction between robot and robot, and also models of interaction between robot and environment.

8. The Project Experience

This project was a great experience in prototyping, working with a customer, third-party libraries, and more. Working on a prototype is notably different in requirements than the typical project that an undergraduate Computer Science is familiar with. For one, the objectives are different. Instead of being assigned specific requirements on functionality, it was up to our team to come up with interesting ideas worth trying. We spent the first term essentially defining our own problem. The amount of time spent doing this part of our project represents its importance. A related experience was working in collaboration with NASA. Since they were our customers, we had to communicate with them about our ideas & progress, and receive feedback on a weekly basis. Learning to ask the right questions and to take advantage of our one hour meetings was difficult.

The other major lesson learned from this project was an understanding of what aspects of a project take the most time. We had not realized that implementation would be so difficult for a program whose functionality we had already defined quite clearly. The main problem with implementation came from working with third-party libraries. While we had taken the time to research all the alternative libraries, the best choices still had their difficulties. Challenges with the neural network training and architecture were not covered in their documentation. And documentation for the FuzzyJ toolkit that is layered on top of the Jess toolkit was woefully inadequate and poorly organized. The Case Based Reasoner framework also lacked sufficient documentation and most progress was made through trial and error.

Ultimately, the project experience was very educational. We learned a great deal about working on larger projects and the importance of timelines. We experienced the entire software development process, from gathering requirements to research, design, development, and evaluation.

8.1 Work distribution

Research during the first term of the project was split between us so that time wouldn't be wasted by both of us having to learn the intricate details of each topic. We both worked on researching robot models in order to generate plausible defects and symptoms. Research on reasoners was split such that we each had a reasoner to research in depth and we would both research rule and fuzzy reasoners as well as means for meta-reasoning.

We worked with Professor Brown during this first term to outline our meta-reasoning strategy including reflex reasoning, rough classification, and sponsor selection. Michael was far more proficient with the Java programming language and the Eclipse development environment so most of the implementation was done by him, with the exception of the CBR (which was Elijah's area of research expertise) and GUI design and implementation. We both worked to debug various areas of the code.

Many sections of the report involved work from both of us including the: abstract, introduction, general analysis of the problem, evaluation, work distribution, and glossary. These sections were edited by Elijah.

8.1.1 Michael's Work

Michael did research on Neural Networks and problem scenarios from the Mars rovers and presented his findings about problem scenarios to Walt at our teleconferences.

He also wrote up our first draft of problem scenarios, listing the problem, descriptions, and related faults. Michael also wrote up many intermediate design documents describing code patterns and techniques to be used.

The second and third term were implementation. Michael implemented the simulation framework in which the reasoners operate. In order to build the framework, he implemented the robots, their network for communication, and the interfaces for communication. He also defined the interfaces for the reasoners and their relations to each other, effectively allowing us to develop the reasoners separate from the simulation framework. He also developed the standard representation of scenarios, problems, symptoms, operating conditions, and more, as part of the frameworks unified data format. This allowed the reasoners to use an internal format appropriate for the third-party library, but allowed the simulation to use a consistent format. Michael implemented the Neural Network reasoner and the Fuzzy-Rule-based reasoner. He implemented their training or knowledge, and did appropriate testing for both.

For the paper, Michael created many of the diagrams in the implementation section. Michael wrote the background research on the Neural Network, and wrote the portions of the Reasoner Tool Selection section about the JOONE, Jess, and FuzzyJ libraries. The Artificial Scenarios section was written by Michael as well. The project experience section was also created by Michael. Finally, the majority of Michael's work on the paper consisted of section 5: the Design and Implementation section. All of section 5, excluding the Case-Based Reasoner and GUI sections, was written by Michael. All un-cited images in this those portions were created by him.

8.1.2 Elijah's Work

Elijah did research on Case-Based Reasoners and Expert Systems. He also worked to produce the initial models of the patient robot.

Elijah handled designing and implementing the GUI for the system. This included getting feedback on the interface from our NASA advisors at meeting and making refinements to the GUI and the way it interacted with the simulation code. The GUI testing and the usability study was also designed and carried out by Elijah. He also implemented, trained, and tested the CBR. This demonstrated the flexibility of Michael's code as the CBR was developed externally and then easily ported into the system.

The work on the report that Elijah did included: editing any joint work portions, most of the background writing (case based reasoners, agent communication language, rule based systems, and fuzzy logic), synthesizing and getting feedback from Walt on our project requirements, the GUI and CBR implementation sections, and the conclusion).

References

Bogaerts, Steven, and David Leake. "IUCBRF: A Framework for Rapid and Modular Case-Based Reasoning System Development." Updated:10 Aug 2005. Indiana University Department of Computer Science. Accessed:4 Oct 2007. <http://www.cs.indiana.edu/~sbogaert/CBR/IUCBRF.pdf>

Dearden, Richard William. "Efficient On-Line Fault Diagnosis for Non-Linear Systems (Diagnostic and Power Port Standard)." Personal Correspondence. Aug 2007.

Friedman-Hill, Ernest. "About Jess 7." *Jess, the Rule Engine for the Java Platform*. 31 Updated: Oct 2006. Sandia National Laboratories. Accessed: 3 Oct 2007 <http://www.jessrules.com/jess/charlemagne.shtml>

Hammond, Kristian J. *Case-Based Planning: Viewing Planning as a Memory Task*, Academic Press Inc, 1989

Jackson, P. *Introduction to Expert Systems. 3rd ed*., Addison Wesley, 1998

Kolodner, Janet. *Case-Based Reasoning*, Morgan Kaufman Publishers Inc, 1993

Kunzle, Philippe. "Vehicle Control with Neural Networks" GameDev.net. Accessed: April 22, 2008. <http://www.gamedev.net/reference/programming/features/vehiclenn/figure1.png >

Leake, David B. *Case-Based Reasoning: Experiences, Lessons, & Future Directions*, AAAI Press/ MIT Press, 1996

Marrone, Paolo. "The Complete Guide: All you need to know about joone" Updated: 17 Jan 2007. The NRC Institute for Information. Accessed: 22 Feb 2008. <http://prdownloads.sourceforge.net/joone/JooneCompleteGuide.pdf?download>

Marrone, Paulo. "Features." *Java Object Oriented Neural Engine*. Updated: 18 Jan 2007. Accessed: 3 Oct 2007 <http://www.jooneworld.com/features.html>

Minsky, Marvin, "Chapter VII Thinking." *The Emotion Machine*. Updated: 28 July 2005. Accessed: 12 Sep 2007 <http://web.media.mit.edu/~minsky/E7/eb7.htm>

Mitchell, T.M. *Machine Learning*, WCB McGraw-Hill, 1997

Orchard, Bob. "Fuzzy J Toolkit for the Java Platform & Fuzzy Jess." Updated: 5 Sept 2006. NRC Institute for Information Technology. Accessed: 3 Oct 2007 <http://www.iit.nrc.ca/IR_public/fuzzy/fuzzyJToolkit2.html>

Orchard, R. "FuzzyJ Toolkit for the Java™ Platform User's Guide" Updated: Sep 2006. Accessed: 22 Feb 2008. <http://www.iit.nrc.ca/IR_public/fuzzy/fuzzyJDocs/index.html>

Punch, W. F., A.K. Goel, and D.C. Brown. "A Knowledge-Based Selection Mechanism for Strategic Control with Application in Design, Assembly, and Planning." *International Journal of Artificial Intelligence Tools* 4.3(1995): 323-48.

Russell, S.J. and Norvig, P. *Artificial Intelligence: a modern approach 2nd ed*, Prentice

Hall, 2002

Stefik, Mark. *Introduction to Knowledge Systems*. Morgan Kaufmann, 1995

Truszkowski, Walt. “What is an Agent?” Personal Correspondence. Sep 2007.

Weiss, Gerhard. *Multiagent Systems: A Modern Approach to Distribute Artificial Intelligence*, MIT Press, 1999

Wikipedia. "Agent Communications Language." *Wikipedia, The Free Encyclopedia*. 28 Nov 2007. Wikimedia Foundation, Inc. <http://en.wikipedia.org/w/index.php?title=Agent_Communications_Language&oldid=174262839>

Winston, P.H. *Artificial Intelligence 3rd ed.*, Addison Wesley, 1993

Woolridge, Michael. *An Introduction to Multiagent Systems*, John Wiley & Sons Ltd, 2002

Yaochu, Jin. “What is Soft Computing?” Updated: Jan 2006. Accessed: 12 Sep 2007. <http://www.soft-computing.de/def.html>

Glossary

ACL – Agent Communication Language

Agent – A software component with processes driven by perception to affect its environment. (See: Truszkowski, Walt. "What is an Agent?")

Agent-to-Agent Communication Module – A component of the Goddard Agent Architecture (GAA) responsible for all communication between an agent and other agents.

ATAC – Agent-to-Agent Communications

Case – A pairing of a problem and a solution used by a case-based reasoner (CBR).

CBR – Case-Based Reasoner

Command – A message sent to another agent to get the agent to perform some task.

Defect – The result of the diagnosis; the hypothesized cause of the symptoms.

Diagnosis – Process of determining a possible cause, defect, for Symptoms presenting in the Patient.

DPP – Diagnostic and Power Port

Environment Context – Data available to the MRDA which describes various data external to the robot (environmental temperature, dust levels, etc.)

Fault – An alert Symptom, for example: a lower power fault occurs when a robot is low on power.

GAA – Goddard Agent Architecture

Meta-Reasoner – The component within our agent that is responsible for determining the appropriate technique for diagnostic reasoning.

Mitigation – A set of actions with the intention of treating the hypothesized defect.

Model – A representation of the Patient including the robot state.

MRDA – Meta-Reasoning Diagnosis Agent

Patient – The robot being diagnosed and treated through the DPP.

Patient Context – Data from the patient robot state that helps to diagnose, treat, or plan treatment.

Response – The Patient's state reaction to treatment. Short for "response to treatment".

Robot State – the actual state of the Patient.

Sensor – A software or hardware component in the Patient responsible for detecting anomalous conditions. These produce what ultimately comes to our agent as a symptom.

Symptom – Data from the robot state that is considered irregular, also watchdog alerts.

Appendices

A. Table of Defect Symptom Details for Training Reasoners

	Power				**Transportation**				**Command and Data Handling**		**Communications**				**Utility**			Reflex	
	Battery Dead from Short Circuit(night)	Battery Dead from Short Circuit(day)	Poor Solar Positioning	Dust on Solar Panel	Broken Wheel	Broken Drive Shaft	Broken Motor	Positioning Malfunction	CPU Overheated	Corrupt Firmware	Directional Antenna Out of Line of Sight	Broken Directional Antenna	Omni-Directional Antenna Out of Range	Broken Omni-Directional Antenna	Broken Camera	Broken Arm	Broken Detector	Return to Base	Drained Battery
	1	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***	***16***	***17***	***18***	***19***
Low Voltage Fault			POS	POS															
High Current Fault	POS	POS																	
Battery 1 Charge	UNDER	UNDER																	UNDER
Battery 1 Temperature	UNDER																		
Angle to Power Output Ratio			HIGH	LOW															
Inefficient Solar Panel Fault				POS															
Low Power Fault	**POS**	**POS**	**POS**	**POS**															POS
Wheel Motor High Temperature Fault						POS													
Wheel Motor Verification Fault																			
Anticipated Movement Fault					**POS**	**POS**	**POS**	**POS**											
Motor Encoder Fault							POS												
Drive Shaft Encoder Fault						POS	POS												
Wheel Encoder Fault					POS	POS	POS												
Electronics Module Temperature	UNDER	OVER							OVER	HIGH									
Electronics Module Low Temperature Fault	**POS**																		
High CPU Temperature Fault		POS							**POS**	**POS**									
Boot Failure Fault										POS									
Increased Idle Processing State Fault										POS									
Directional Antenna No Response Fault											**POS**	**POS**							
Directional Antenna Verification Fault												POS							
Omni-Directional Antenna No Response Fault													**POS**	**POS**					
Omni-Directional Verification Fault														POS					
Arm Verification Fault																POS			
Camera Verification Fault															POS				
Detector Corrupted Data Fault																	POS		
Camera No Response Fault															POS				
Return To Base Flagged																		1	
Lunar Daytime		1																	