University of Nevada, Reno

Development of an Autonomous Rover for the Nevada Student Satellite Program

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science in
Computer Science

by

Pablo A. Rivera

Dr. Monica N. Nicolescu/Thesis Advisor

Dr. Eric L. Wang/Co-Advisor

Dr. Jeffrey C. LaCombe/Co-Advisor

August, 2007

We recommend that the thesis
prepared under our supervision by

Pablo A. Rivera

Entitled

Development of an Autonomous Rover for the Nevada Student Satellite Program

be accepted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

Monica Nicolescu, Advisor

Mircea Nicolescu, Committee Member

Eric Wang, Committee Member

Jeffrey LaCombe, Graduate School Representative

Marsha H. Read, Ph. D., Associate Dean, Graduate School

August, 2007

Abstract

This thesis presents an approach to designing and building an autonomous rover for the **Nevada Student Satellite** (NevadaSat) program, to compete in the annual **ARLISS Come-Back** competition. This competition begins with a rocket launch and payload deployment. The payload must then be returned to a predetermined location without a human operator. The main challenges of the project consist of the hardware and software constraints. From the point of view of the hardware, the robot had to be built with a budget of $5000 and a small team of researchers. With the software, the challenges were to provide a robust and flexible control architecture which would enable long-term autonomous navigation for an outdoor robot. The solution proposed in this thesis is a simple, flexible and modular architecture, both in hardware and in controller software. The controller design allows for sensors and actuators to be added, deleted, or replaced without affecting the entire controller. The controller configures itself according to available resources and for various tasks; it can run either an entire mission or various phases of it without being rebuilt. The Autonomous Rover is an exercise in the development of a robotic vehicle to perform relatively sophisticated tasks at a low cost, and with limited resources. This thesis presents results obtained from field experiments demonstrating the capabilities of the robot.

Acknowledgments

Thanks go to Dr. Jeffrey LaCombe, Dr. Monica Nicolescu, and Dr. Eric Wang for their continued interest in NevadaSat and participation in the ARLISS Event. I would also like to thank all of the people who helped make the Rover Project possible, including Travis Fields, Guillermo Larios, Blake Poe, and Palkin Zed.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 NevadaSat and the ARLISS Event

The NevadaSat Program is a program based at the University of Nevada, Reno, and operating throughout the Nevada System of Higher Education (NSHE).

> "The objective of this program is to expose students in Nevada to aerospace science and technology through hands-on experience and participation in a wide range of high-impact activities. At present, this is done through two programs (BalloonSats and CanSats)." [Neva07]

CanSats are the focus of the ARLISS event for the NevadaSat Program. The ARLISS (A Rocket Launch for International Student Satellites), is an event held in Black Rock Desert, Nevada once a year usually in September.

> "The ARLISS program is designed to provide an educational experience to students in the design, flight and data analysis of a space experiment. This program is to prepare students for an exciting, technical challenge that may lead to launching space experiments into low earth orbits and beyond." [Arli07]

The event involves the design and deployment of a hobbyist rocket, reaching an altitude of about 10,000 ft A.G.L. (above ground level), or higher, and its payload, the **CanSat**. [LaWa07] The CanSat is a soda-can-sized (thus the name), payload containing a GPS receiver, radio modem, and other instruments. The ARLISS event includes a competition, the "Come-Back Competition," offering a prize to the first team to build some type of autonomous solution to get the CanSat back to a pre-determined flag location with no human intervention. The Come-Back Competition has not been won by any team as of ARLISS 2006.

A few different strategies are used by the participating teams:

[1] Use a larger CanSat (coffee-can size) and have it **drive** back

[2] Use a larger CanSat (coffee-can size) and have it **glide** back

[3] Use the smaller CanSat (soda-can size) and have some type of robotic vehicle retrieve it, then **bring** it back

Most teams choose option 1. The NevadaSat team chose option 3. Each involves its own unique and difficult challenges. For option 1, the challenge seems to mostly involve scale; the small autonomous vehicle must fit inside a coffee-can-sized payload, yet be sufficiently strong to handle the stresses of flight and the landing. As was demonstrated in ARLISS 2006, the small robot may have to do more than just GPS/compass navigation – it may be required to perform obstacle avoidance, a more difficult task considering the smaller platform. With option 3 a robot of much greater size and weight can be deployed for the task, providing greater processing power and room for more devices. At the same time, the distance that must be traveled will be greater and the CanSat will have to located before it can be retrieved.

1.2 Previous Years

2006 is the 5th year that rocket(s) were launched by NevadaSat during the ARLISS containing a CanSat payload. Until 2004 tele-operated vehicles were used in the **Come-Back Competition** to investigate the requirements of the challenge. Much was learned about vehicle design, power demands, batteries, motors, and general navigational issues.

These vehicles were also fitted with a PC-104 board for telemetry, video camera (for live video) and radio transmitter, as well as an arm to grab and retrieve the CanSat.

2005 marked the first year for NevadaSat to participate in the ARLISS with a Rover that would attempt to perform the **Come-Back Competition** autonomously. Many successes were realized during testing in terms of autonomous navigation using GPS and compass, as well as deployment of a highly effective arm design. The vehicle underwent many transformations; the final design was very similar to the one used with the 2006/2007 versions of the Rover and was capable of handling the terrain of the Black Rock over long distances reliably. The computational platform consisted of a Parallax Javelin Stamp Module which uses a Ubicom SX48AC microcontroller operating at 32 MHz with 32 KB RAM. The module could be programmed in Java but was not capable of booting an operating system and thus could run only single-threaded applications. Indeed this platform could be considered an embedded platform in that it imposed severe restrictions in terms of computational resources. It was coupled with a Floating Point Math Coprocessor to perform any floating-point calculations, since these could not be performed on-board the Javelin.

All-in-all, the 2005 Rover had limited success in completing the tasks of the **Come-Back Competition.** The only fully successful test that was conducted placed the CanSat at a distance of about 132 feet from the Rover. The Rover was able to navigate to the CanSat, pick it up, and drive to a pre-determined flag location for a total distance of about 218 feet. Among the challenges left to address was the reliability of blob-tracking during the

"short-game" phase; the color for the parachute had to be chosen very carefully as the tracking was not tolerant of changing light conditions. During the ARLISS, only one launch was made, and the Rover was not able to retrieve the CanSat because the wind was too strong, blowing the CanSat several miles from the launch area. Further launches were canceled due to inclement wind and weather. Finally, the vehicle chassis was damaged severely during transport after the event, and the unit as a whole was retired.

1.3 Our Approach – Overview

1.3.1 Hardware

After the 2005 ARLISS we redesigned the entire Rover, including the vehicle chassis and drive system, electronics, computing platform, and software (Figure 1-1). This new design is the one being used with the Rover today. Since the task remained the same, many of the lessons learned from the 1st autonomous generation were applied to the design of the second. The overall idea was to make components stronger, more reliable, and more versatile. For starters, the new chassis design used stronger, easily replaceable aluminum modular framing components. This made the chassis lighter and provided mount points for the super-structure, drive and suspension systems, with little need for machining and no welding. In this way a relatively advanced chassis design could be constructed quickly, and with few resources. In addition, it is not too difficult to replace damaged parts or modify the existing design because the chassis is so modular.

The drive and suspension systems provide 4-wheel independent-suspension skid-steering, which with the right wheels give the Rover both a high top speed (~10 m.p.h.), and good

maneuverability on the smooth flat terrain of the playa (dry lake bed), found at the ARLISS site. Each wheel is driven by an electric motor. The entire drive system is powered by several sets of NiMH batteries wired through a fused and metered main

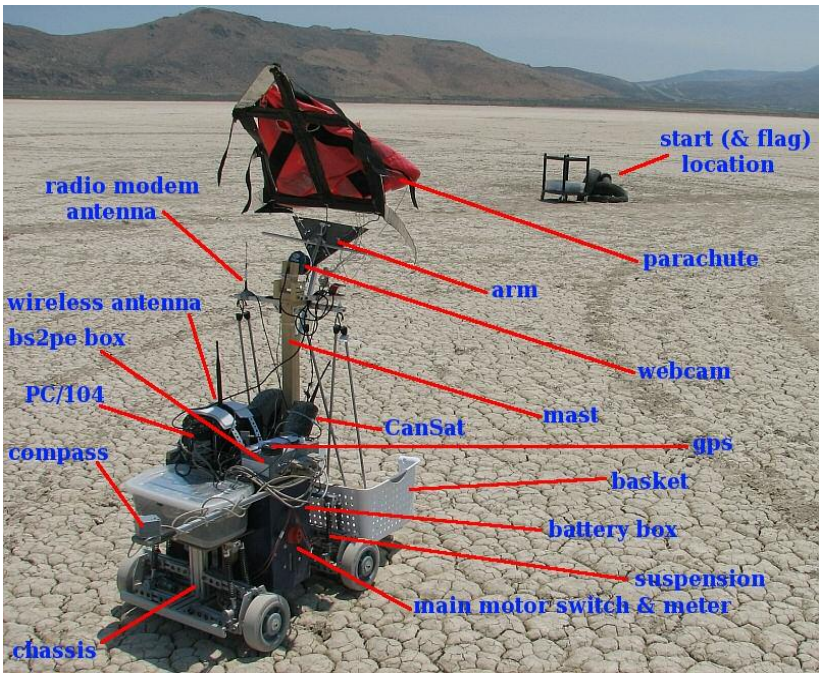

Figure 1-1 – Rover Hardware Component Layout (2007)

board. Testing revealed that enough battery packs could fit on board the Rover to power it during a mission spanning a distance of 2-4 miles. A RoboteQ ax2550 speed controller provides an RS-232 interface to control all 4 motors on 2 channels and is tolerant of very high current draws up to over 100 amps (>250 amp surge). The drive and electronics power systems were kept entirely separate to eliminate noise and the chance of harmful spikes or surges being passed to sensitive components. As it turns out the drive system was rather noisy – enough to disrupt wireless communications with the Rover computer during testing, so capacitors were fitted on the motors to reduce this noise.

There was some debate as to whether we could have used either existing outdoor robots, or retro-fitted some type of gas-powered off-road vehicle with a **drive-by-wire** control system. The final decision was made both due to the higher cost of these solutions and the increased difficulty in implementing drive-by-wire given the resources available to our small team and budget. Also, the unique nature of the task required a more specialized solution than most existing outdoor robots could provide. Even these would have required some retro-fitting and enhancements. These considerations would be repeated with many of the other design decisions, including the software system.

The computational system can be described as consisting of 2 main components: the central processing platform, and the device I/O system (sensors and actuators). The main actuator was the drive system, but there was also a 1-degree-of-freedom arm that is used to physically grasp the CanSat and parachute once it has been found. A summary of sensing and actuating devices is shown in Table 1-1. Sensors consist of a video camera (Creative Labs), gps receiver (Garmin GPS18), electronic compass (Devantech cmps003), and radio modem (from MaxStream). An ultrasonic ranging system was constructed and a driver written, but it was not fully functional by the writing of this thesis. Also an rfid reader/writer is in the works. The specific role of each of these sensors for the controller will be described in more detail later. The processing platform is a PC/104 computer running a custom Linux build for its operating system. The PC/104 uses a disk-on-chip for permanent storage, has enhanced I/O capabilities as compared to the average desktop or laptop in its price range, can run reliably from batteries, uses very

Manufacturer	Model	Role	Notes
RoboteQ	ax2550	Motor Controller	Dual 120A per Channel 12V to 40V DC operating voltage RS-232 or PWM interface
Garmin	GPS-18	GPS Receiver	12 parallel channel, WAAS-enabled 12-volt DC input, RS-232 interface
Creative Labs	Web cam Pro	Video Camera	resolution: 640 x 480 USB 2.0/1.1 Adjustable focus lens, auto exposure
80/20	T-Slotted Aluminum Framing	chassis structural components	“Industrial Erector Set,” strong & light
Diamond Systems	Athena pc/104	On-board main computer	VIA Eden 660MHz Processor, 128MB RAM 4 RS-232 serial ports, 4 USB PS/2 keyboard/mouse ports
Tri-M	HESC104	PC/104 Power Supply	6V to 40V DC input range 5,12 volt DC (60 watt max) output Extended temperature: -40°C to +85°C
Parallax Inc.	BASIC Stamp 2pe Module (bs2pe)	Processing and I/O for peripheral sensor devices	8 MHz Processor Speed, 38 Byte RAM I/O Pins: 16 +2 Dedicated Serial
Devantech Ltd.	CMPS03	Magnetic Compass	Resolution - 0.1 Degree I2C Interface, SMBUS compatible, PWM
MaxStream	9XStream 900 MHz RF	Data Radio Modem	Outdoor line-of-sight Range: Up to 7 miles Power: +5 VDC 50 mA receive, 140 mA transmit
Innovation First, Inc.	VICTOR 883	Speed Controller (for arm motor)	PWM (Pulse Width Modulation) Interface 1 channel

Table 1-1 – Rover Hardware Components (2007)

low power, and has many other features which make it ideal for the Rover. As will be explained later, processing requirements are kept low, so we could run a complete mission on a single 600 MHz processor with 128 MB of RAM.

In addition to the PC/104, a separate processor unit housed 3 circuit boards and the Radio Data Modem. The 3 boards contained custom circuits that used Parallax bs2pe microcontroller modules to interface with devices that used protocols not supported directly by the PC/104. In addition to providing protocol translation, the bs2pe runs simple BASIC programs that poll the devices and make the low-level details of the devices transparent to the PC/104. For example, the arm control board accepts one of three ASCII text commands: "1," "2," or "3," over RS-232 to move the arm up, down, or

Device	Native Protocol	Protocol implemented	Used in current Rover
CMPS03	i2c	RS-232	Yes
VICTOR 883	PWM	RS-232	Yes
Parallax PING)))	PWM	RS-232	No

Table 1-2 – bs2pe Peripheral Device Interface Box

report its current status. These boards are described in Table 1-2. Using the bs2pe modules in this function simplified development of drivers, and was initially a requirement since the first PC/104s we used **did not** support data acquisition so there was no way to access devices through digital I/O, pulse-width modulation, or i2c. The final

PC/104 that was used, and the one currently on the Rover, **does** support data acquisition, so drivers could be written to talk to these devices directly. Another alternative would be to use one of the increasing numbers of electronic compass devices, servos, and ultrasonic rangefinder arrays that support RS-232 or USB interfaces natively.

1.3.2 Software

The software system can also be described as having 2 main components – the operating system and controller applications. Gentoo Linux was chosen for the operating system because it is widely used and well-documented, because of the budget (no licensing costs), and because of its almost limitless configurability and maintainability. Gentoo is a self-described “metadistribution;” it does not provide a default binary build, so nearly any aspect of the operating system can be modified. Thanks to Gentoo's portage system of software management, building and installing a myriad of open-source packages is fairly straight-forward, for anything from word processors, to robot server applications, to device drivers. On the down side, a common challenge with any Linux system is that many times new devices do not have drivers because many manufacturers do not provide them (although the situation is improving). This limited us to a subset of all available devices, but in the end there were plenty to choose from and a final sensor/actuator suite was assembled that did the job nicely.

The Rover's operating system has a very small footprint both on disk and in memory. Although it has a desktop environment installed and available (fluxbox), it is not loaded by default since it is not used during an autonomous mission. It was also possible to make

most of the filesystem read-only and residing mostly in memory by using an initrd, or initial ramdisk and configuring the mounting of partitions containing essential parts of the system to be read-only. This was not only important to avoid corruption during unexpected power failures, but also minimized excessive write cycles to the flash disk while improving I/O operations – memory is usually much faster than persistent storage media. A minimal kernel loads driver modules only as they are necessary. New drivers can easily be built as modules and loaded/unloaded as is necessary. Even though the PC/104 provides a platform similar in many ways to a PC desktop or laptop computer, the special requirements on the Rover benefited from using a somewhat embedded system.

Various software packages provided essential functions during controller development. Among these were the Apache web server, remote synchronization utilities – nfs, rsync, and a chroot build environment managed through bash scripts. Apache provided an easy light-weight avenue for the Rover to transfer dynamic data to and from the monitoring station (laptop) during development; device configurations could be updated, and large files such as images from the camera transferred with short, simple scripts. At the same time, the data is human-readable and accessible through any web browser. It provided a graphical, user-friendly way to monitor the Rover remotely as it drove around without the overhead usually associated with developing conventional GUIs (Graphical User Interfaces). nfs and rsync were used because the Rover software was developed and built mostly on the laptop host machine. Using a chroot environment allowed the operating system to be built and maintained on the host laptop, then nfs provided a mechanism to

transfer gigabytes of data across the network to the PC/104. rsync was used for quick, reliable updates of the operating system. The controller software was also built and tested to a great extent on the laptop thanks to the Stage simulation environment. Finally, the code base for the controller was managed with subversion, a powerful yet very approachable version control system.

The other main software component, and subject of the remainder of this thesis, is the controller. The controller was written in C/C++. It is closely integrated into the Player/Stage library (see chapter 3.1), it runs on top of – it not only depends upon it in its design and function, a few new Player drivers were written to support some of the devices on the Rover for which drivers did not exist. In keeping with the modular nature of the rest of the Rover, the controller is designed to utilize whatever devices are available according to the task it is given. As will be explained later it uses the Player/Stage client/server architecture to operate through a set of interfaces rather than specific devices. In this way, any sensor or actuator could be replaced with a different device so long as the replacement can do the same job.

The controller architecture is mostly reactive, but also has deliberative components. In this way it could be described as a hybrid architecture although it is different than most hybrid architectures. It differs in that there is no hierarchy or subsumption of reactive and deliberative components. There is no central sequencer or deliberative process – the deliberative processes work at the same level as the reactive ones in controlling the overall behavior of the robot. The controller also uses a generalized organization that

configures itself according to configuration files, command-line options, and available resources during initialization.

Overall, the modular, flexible nature of the hardware and software system made it possible to develop the complete solution in an incremental manner. Device failures along the way did not force major changes in software. It was possible to test partial instances of the mission task with the same controller that was used to run a full mission. The design leaves plenty of room to add additional components, or replace hardware and software components with better ones without changing the main structure.

Chapter 2 Related Work

2.1 Projects – DARPA Grand Challenge

The DARPA Grand Challenge (DGC) provides a robotic challenge that is similar to that of the ARLISS Come-back Competition, but also very different. Among the similarities:

- required a rugged, maneuverable off-road vehicle
- required GPS way-point navigation
- required some form(s) of obstacle avoidance
- required robust operation in an unstructured outdoor environment

One of the main differences is in the scale and scope of the event. Most participants, and all winners of the 2006 DGC, were well-funded and had teams composed of dozens of members. The task involved in the DGC is in many ways more complex than the Come-back Competition because it requires road-following in rugged terrain, and much higher

speeds. The DGC is also different in that the task is less sequential, there are no well-defined phases that must be performed in order, other than the course itself.

It is useful to compare the strategies used by participants in the DGC, both for teams that completed the challenge and those that did not. Most teams used inertial sensors, LIDAR, drive-by-wire, and an emphasis on machine vision [ReMa06]. In addition, controller software architectures were highly complex, using deliberative components and processing algorithms with requirements that could not be met by a single-processor embedded system. Some teams used parallel computers and redundant sensors, fully utilizing the space and power supplies available on the off-road vehicles they were fitted on. Finally, many of the winning teams realized the importance of using interactive Graphical User Interfaces (GUIs) during development, some with data playback capabilities [BrBr06].

2.2 Software Architectures

The three main paradigms for robotic control are reactive control, deliberative control, and hybrid control. Each has its advantages and disadvantages, depending on the nature of the task a robot is to perform.

2.2.1 Deliberative Control

Deliberative Control is modeled after classical Artificial Intelligence (AI), and was the dominant paradigm with early robots. It requires the robot to formulate a plan about its actions based on sensor data and an internal representation of the world based on

symbols. Robots utilizing this form of control were known for being slow and unreliable [Arki98]. It is very difficult to write a purely deliberative controller that can accomplish anything but simple tasks in real-time, and even more difficult to make it tolerant of noisy, unstructured environments. For these and other reasons, purely deliberative control has been abandoned as the main focus of robotics research.

2.2.2 Reactive (Behavior-based) Control

The first proponents of reactive control architectures aimed to eliminate the drawbacks of deliberative control by not using deliberation at all. The idea is to map sensor data directly to actuator commands in a reflexive manner as quickly as possible. Among the benefits to this approach are its low computational overhead and real-time operation [Broo91]. By combining fairly simple reactive components (also referred to as behaviors), it is possible to achieve more complex tasks. Many examples of successes exist for this type of control, which is based partly on observations of natural behavior in animals [Mata98]. Combining simple reactive components that are closely linked to sensor data from the environment can result in robust, modular controllers that are well suited to unstructured environments [Stee90].

Among the drawbacks of purely reactive systems is the awkwardness encountered in trying to implement highly sequential tasks. Due to their nature, some applications may require some form of deliberation. Among these are temporal sequencing and planning.

2.2.3 Hybrid Architectures

In an effort to combine the benefits of both reactive and deliberative architectures, most modern robotic controllers use what is referred to as a hybrid architecture. There are many approaches to performing the integration of deliberative and reactive components [Arki98]. Among them are arbitration and layering. The Distributed Architecture for Mobile Navigation (DAMN) uses behaviors consisting of both reactive and deliberative components at the same level of control, then fuses their actuator commands using a voting system. The fusion is performed by "arbiters", which accept votes from the behaviors, at different frequencies, and uses them to select the next discrete command to be sent to the particular actuator that they control. One of the issues with DAMN is in performing fine-grain, continuous control:

> "Although the internal command representations used by the arbiters are discrete, the inputs to them from the voting behaviors need not be, and indeed the interfaces allow for the specification of continuous voting functions. This introduces the questions of whether and how accurately these inputs need to be reconstructed for the sake of precise and stable control, and how to best effect it. A related question that arises in both continuous and discrete systems is whether and how smoothing should be performed. " [Rose97]

In layering, deliberative and reactive components are separated into layers of the control hierarchy, where a deliberative planner operates at a higher-level coordinating the actions of lower-level schemas or reactive components [Arki98].

What constitutes a hybrid architecture and what does not is itself contested among researchers. One issue with purely reactive architectures is the difficulty in integrating any form of temporal sequencing, and especially, representation for the purpose of action

planning. Some success has been demonstrated with integrating maps, for example, directly into a reactive architecture with minimal use of symbolic representation [Mata92]. However, most of these have been in structured indoor environments. It also seems that any form of symbolic representation, and the basing of decisions on it, fits the definition of what deliberation is, regardless of how fast or minimal. Since control based on deliberation is usually slower and more difficult to implement reliably, a design principle with many hybrid architectures is "React when you can, Plan when you must".

Chapter 3 Controller Software

3.1 Player/Stage

The Rover software controller is built to run on the Player "robot server" [Play07]. Player is a client/server architecture that provides a level of abstraction for devices, allowing a controller (the client), to see sensors and actuators as consistent interfaces regardless of the actual device being used (one of the services provided by the server). Player aims to "to follow the UNIX model of treating devices as files," [Play07] in that the driver hides the low-level details of device I/O and provides a consistent read/write interface to applications, with multiple simultaneous connections. With the Rover, Player allows us to build a multi-threaded application where the controller and device drivers are on separate threads much more easily – without the synchronization challenges that can be encountered in building multi-threaded applications in C and C++. Using one configuration file, Player will run with any combination of available devices in any available configuration. The controller can then connect to it through a set of proxies,

discovering which devices will be available and in turn – which devices will be used. Player proxies are handles to the device interfaces that Player provides. Player also makes it possible to write new drivers for unsupported devices as plug-ins. At least half of the devices used on the Rover required a plug-in driver to be written but once the drivers were written, Player provided a consistent set of interfaces. Because of this approach any of the devices could be replaced by an equivalent device transparently.

One example of this was with our earlier blobfinder device, the CMU Cam2, which was replaced by a Creative Labs webcam. This change did not require any changes to the controller. All that was required was the Player driver plug-in. The opencv_blob driver that we developed can use any camera that is compatible with Video4Linux. An additional requirement is that the camera support Phillips Web Cam (pwc) extensions. OpenCV libraries are then used to interface with Video4Linux and support a rich set of image processing functionality.

The opencv_blob driver uses a novel adaptive method to perform blob-tracking in changing light conditions. A small patch of the same fabric used on the parachute is physically placed on a rigid mount in front of the camera so that the patch will always appear in the lower-right corner of the camera's field-of-view. The patch is likely to be very close to the color of the parachute that is being tracked, so it can be actively sampled by the driver to get a more accurate range of color thresholds to sample in order to blob-track the parachute. We developed a tool to perform initial calibration for the driver. The tool provides a graphical user interface (GUI) which allows the user to specify settings

for gain control, image processing transformations, and the coordinates for the sample patch area. The tool runs on a laptop having a network connection to the Rover. These configurations are then saved to a file and made available to the Rover controller over the network for it to load when it initializes.

Besides its device server capabilities, another main feature of Player that was instrumental to the development of the Rover is its sister application, Stage.

> "Stage simulates a population of mobile robots moving in and sensing a two-dimensional bitmapped environment. Various sensor models are provided, including sonar, scanning laser rangefinder, pan-tilt-zoom camera with color blob detection and odometry.
> Stage devices present a standard Player interface so few or no changes are required to move between simulation and hardware. Many controllers designed in Stage have been demonstrated to work on real robots." [Play07]

Since Stage simulates data for the same devices that are supported in Player, it is possible to test a controller in simulation before taking it into the real world, with minimal changes to the code. It is possible to develop the software even while the real robot is being built or repaired. This allows hardware and software to be developed incrementally and in parallel. For example, the Rover's chassis was disassembled in order to enhance the suspension, fairly late in the software development process. Stage allowed components of the controller to be coded and tested in simulation, then deployed with few changes during real testing when the hardware upgrades were complete. There were several instances when critical hardware components failed and had to be replaced. Without Stage, days or weeks would have been lost waiting for the parts to arrive.

Aside from the conveniences provided by its abstractions, Player/Stage influenced the way the controller was designed; it lent itself well to the Behavior-based control paradigm used to develop the controller. The interfaces provided by Player could be composed into primitive behaviors, in much the same manner that the primitive behaviors were composed into abstract behaviors, then in an overall resultant behavior in the main controller. Primitive behaviors, like Player drivers, could be found or developed as new features were added. Since Player configures itself as a "device proxy server" when it initializes according to a configuration file, the main controller reads that same file and a few other options, then configures itself as a "robotic task server" according to the resources it has available.

3.2 Architecture

3.2.1 High-level Overview

The controller is based on a Behavior-based Control (BBC) architecture. The controller's primitive and abstract behaviors function on different levels of organization. As is implied by the term "primitive," primitive behaviors are the building blocks, smaller units of functionality, that can be combined to form the abstract behaviors. The abstract behaviors are less well-defined but combine simpler elements to result in achieving overall goals [NiMa02].

The controller reads in configuration options as specified in the Player configuration file, from the command line that invokes it, from a small set of compiler options, and from a few other files (Table 3-1). The configuration options tell it which resources it should

expect to have available – including hardware devices as provided through Player proxy interfaces, any hardware not accessed through Player, data about waypoints or maps, and the primitive behavior weights for each abstract behavior composition.

Configuration File	Blob	Motors	GPS	Compass	GPS Beacon	Sonar	Arm	A*
"opencv_blob.cfg"	X						**	***
"blob.cfg"	X	X					**	***
"gps_cmps.cfg"		X	X	X			**	***
"gps_cmps_beacon.cfg"		X	X	X	X		**	***
"all.cfg"	X	X	X	X	X		**	***
"stage.cfg"	*	*	*	*		*	**	***
"stage_beacon.cfg"	*	*	*	*	****	*	**	***

* implemented by Stage (simulated data)
** Command-line option
*** Compiler option
**** implemented through a fifo (with simulated data)

Table 3-1 – Player Config File Options and Controller Configuration

During initialization, the controller declares the Player proxies then declares primitive behavior objects, passing to them pointers to the proxies and configuration options that pertain to them. It uses a set of inference rules to decide where data for pre-conditions will come from, and what tasks it will be performing. Each abstract behavior has a file containing weights for each of its primitive behaviors, which also defines the composition of primitives for that abstract behavior. The controller gives the abstract behaviors pointers to their primitive behaviors, and defines the preconditions that will activate them.

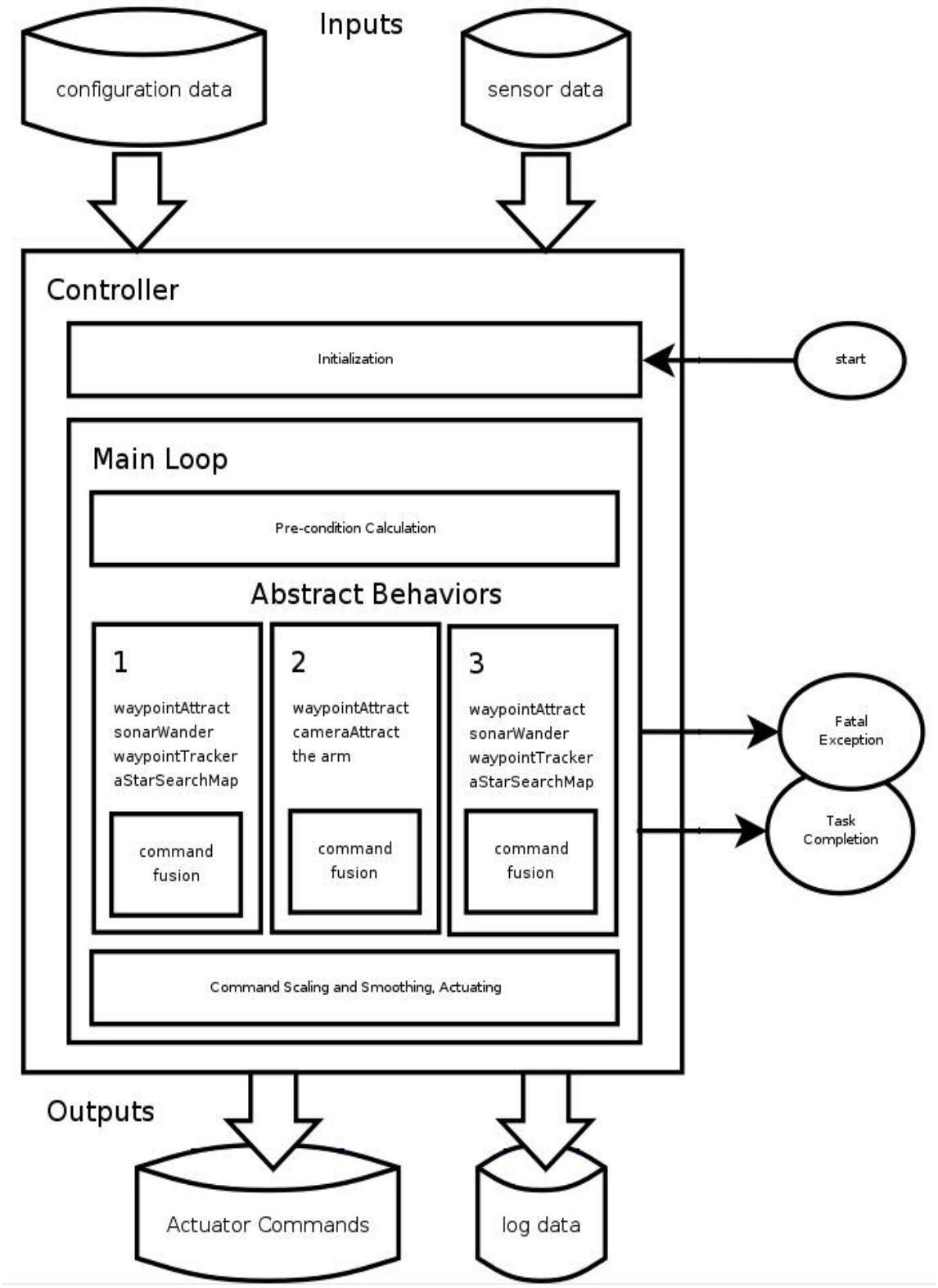

Figure 3-1 – Rover Controller High Level Architecture

Once initialization completes, the controller enters the main loop (Figure 3-1). At the top of the loop, the set of preconditions is checked and the abstract behavior whose preconditions are satisfied runs. If no abstract behavior can run, the default action is to terminate with an error. If multiple abstract behaviors could run, only the first whose preconditions are satisfied will run. It is important to design preconditions so that they will be mutually exclusive and represent the entire state space to avoid undefined behavior.

There is one special precondition which can cause the main loop to terminate – finished. When the main loop is terminated due to that precondition, the controller exits normally and declares the task has been completed. At the bottom of each loop, the actuator commands that were output by the abstract behavior that just ran are adjusted according to the actuator they are running on. The function, adjustActuatorCmd(actuator_cmd) insures that the command that is issued to the corresponding actuator will be within valid ranges and smoothed to avoid abrupt, potentially damaging movements.

Each abstract behavior is responsible for calling its primitive behaviors and fusing their commands together. This is accomplished through vector addition, weighting, and normalization. An actuator command for the drive motors will be a trajectory vector expressed as (magnitude, direction). Magnitude is a scalar value within a bounded range; for example -1 to 1 (max reverse to max forward). The abstract behavior calls the primitive behaviors one-by-one, multiplies their output vector(s) by their scalar weight

value, and merges the result into a resultant vector using vector addition. If an output has a magnitude component of zero (0) it is not merged into the total vector since it is assumed that the behavior has no real output to report. At the same time, the abstract behavior accumulates the total vector magnitudes using scalar addition. At the end it normalizes the total vector by dividing it by the total magnitude (after checking for divide-by-zero exception). This is so that the weights will affect each primitive behavior's output relative to the other behaviors, while keeping the total magnitude within valid bounds (in this case -1 to 1).

Here is an example of using this method: say an abstract behavior is composed of a blob-tracking primitive behavior given a weight of 2, and a sonar-avoid primitive behavior given a weight of 1. Each behavior would provide an output vector magnitude for the drive motors between -1 and 1. During one iteration the blob-tracking behavior outputs a vector (1, 45 degrees) and the sonar-avoid behavior outputs (0.5, 0 degrees). During command fusion the abstract behavior would perform the following calculations:

1. multiply output vectors by scalar weight values:

   (1, 45 degrees) * 2 = (2, 45 degrees)

   (0.5, 0 degrees) * 1 = (0.5, 0 degrees)
2. merge vectors using vector addition:

   (2, 45 degrees) + (0.5, 0 degrees) = (2.4, 36.5 degrees)
3. accumulate total magnitude of vectors using scalar addition:

   2 + 0.5 = 2.5

4. normalize total vector (from step 2), by dividing by total magnitude (from step 3): (2.4, 36.5 degrees) / 2.5 = (0.96, 36.5 degrees)

The resulting vector, having a magnitude of 0.96 and direction of 36.5 degrees demonstrates that since the blob-tracking behavior was given a bigger weight value, it had greater influence on the resulting vector, yet the magnitude of the resulting total vector is still within bounds ($0.96 < 1$).

Another important function of the controller is that it implements extensive logging. With robotics applications this is especially important. The causes of failure can be more difficult to trace than with applications running only in a software domain, indoors, on stable predictable hardware. Data gathered by sensors on an outdoor robot in varying conditions is likely to show the effects of noise and contain frequent errors. Add to this the relatively large state space the robot and its software operate in, and it can become almost impossible to track bugs without well-organized logs of as much relevant data as possible. The Rover's software controller logs sensor data, code branches (if..then, switch statements, etc..), conversions and interpretations of sensor data, configurations and inferences made during initialization, behavior outputs, final actuator commands; anything that will affect the outcome of task execution is logged. Logging data is organized according to which function and class produces it. For example, output generated by the main controller function is prefaced with “controller:”. Output from the cameraAttract behavior is prefaced with “cameraAttract:”. This is done to make it easier to analyze later since it can be determined which section of the code was doing what

during any phase of the task. Logs were used countless times to discover both subtle and obvious bugs and to verify whether fixes were successful. Late in the development of the controller it was discovered that Player/Stage offers tools to help robotics developers with producing, analyzing, and even replaying log data. One improvement would be to see if the controller's logging functionality can be integrated with this to make it even more useful.

3.2.2 Reactive and Deliberative Components

3.2.2.1 Primitive Behaviors

In the controller, behaviors are organized hierarchically, with “abstract” behaviors being compositions of “primitive” behaviors. The primitive behaviors are objects which provide some type of service for the controller. In this way there are like “agents” that can, given the necessary inputs, calculate an output without knowing anything about the other behaviors or the main controller. For the remainder of this thesis, primitive behaviors may also be referred to interchangeably as “agents”. Some of them provide actuator commands, some provide filtering or monitoring of sensory input, some provide both. These primitive behaviors are then combined to form abstract behaviors. Both types have preconditions that decide whether they will be active or not. In the Rover controller primitive behaviors can, and are, simultaneously active. So it is possible for the preconditions of many of them to be satisfied concurrently. The abstract behaviors, however, have preconditions that are mutually exclusive, ensuring that only one will be active at a time. Each abstract behavior has exclusive control of the actuators while it is

active, but as will be explained later, they do not have to follow any particular order of execution.

The primitive behaviors we developed are called **waypointAttract**, **cameraAttract**, **sonarWander**, **waypointTracker**, **aStarSearchMap**, and **armPickup**. Although the names they were given aren't exactly consistent, they provide a good idea of what they do. Each one has a set of preconditions, inputs, and outputs. Some are completely stateless, while others hold an internal state.

Here are some brief descriptions:

- **waypointAttract** – use position data to calculate a trajectory toward a waypoint and an interpretation of the current tracking "state"
- **cameraAttract** – use blobfinder data to calculate a trajectory toward a color blob and an interpretation of the current tracking "state". In addition, it will attempt to search for the color blob if none is initially perceived, using its own searching procedure
- **sonarWander** – use ultrasonic data to calculate a trajectory towards open areas
- **waypointTracker** – monitor GPS data from a "GPS beacon" – report the current data state and best estimate of its location
- **aStarSearchMap** – take a start and end location (in GPS coordinates), and plan a path avoiding obstacles defined on a map
- **armPickup** – use the arm to pick up the CanSat and parachute

The controller is structured such that each primitive behavior can be completely rewritten without affecting the controller as long as its interface remains the same. New primitive behaviors can easily be added and integrated into new or existing abstract behaviors. In general, behaviors providing output for the drive actuator (motors), have a method called trajectory(). Those providing output about sensor data states will have methods such as getData(), flagIsSet(), dataStatus(), and lastKnownData(). Inputs consist of configuration requests and sensor data coming from one or more Player proxies. The proxies used with the controller include **SonarProxy**, **BlobfinderProxy**, **Position2dProxy**, and **GpsProxy**. All behaviors use the following strategy regarding proxies: if the pointer to the proxy that is passed to them is nil, they will not attempt to use the proxy as an input, but will report a zero actuator command or error data condition.

The following is a more detailed description of the algorithms used in all behaviors:

waypointAttract

This behavior converts compass heading data and GPS coordinate data to coordinates on a 2-dimensional grid where the x-axis is UTM Easting (increasing left-to-right), and the y-axis is UTM Northing (increasing bottom-to-top). Zero heading is East, PI/2 is North, etc.. Given this current location and coordinates for a waypoint, the pre-condition **dist_to_waypoint**, and local variable **heading_to_waypoint** are calculated. **heading_to_waypoint** is compared to the current heading to calculate a trajectory

direction component. Trajectory magnitude is a discrete value stop (0) or full-speed (1) depending on whether a trajectory can be calculated.

cameraAttract

This behavior takes blobfinder data for the first current blob (only one blob is tracked at a time here), and uses its centroid coordinates and area to calculate a trajectory. **cameraAttract** is not state-less. It keeps history items to calculate a current "tracking state:"

- NONE – there is no valid blob data to report
- REACHED – there is blob data to report and it meets or exceeds the criteria for "blob reached". This state would indicate that the Rover is as close as it can get to the CanSat (assuming it is the blob being tracked), using just blob tracking
- ACTIVE – there is data to report. This could be fresh data directly from the blobfinder proxy, or data retrieved from the history (last fresh reading)

The **cameraAttract** history operates as follows: each time fresh data is read in (ACTIVE), it is saved as **last_blob_data**. A counter for the number of consecutive no-data available readings is set to zero. The first time that no data is available from the proxy, the counter increments and data is retrieved from **last_blob_data** and reported as the current data available (ACTIVE). Each time this happens the counter continues to increment until it exceeds a pre-defined threshold, BLOB_ZERO_EXCEEDED. From this point on no data will be reported (NONE), until fresh data is again available from the

proxy. Another component of the history is a pre-defined threshold for reporting a REACHED state. Since the REACHED state is a very important flag for the abstract behavior that uses it, **cameraAttract** wants to have some certainty that the blob actually meets the criteria for "reached" and that the data is not a temporary error. It will wait until a counter **blob_num_times_reached** has exceeded the BLOB_REACHED_THLD before reporting a REACHED state otherwise reporting just ACTIVE.

As was mentioned previously, when it is called, **cameraAttract** will in the absence of valid blob data, go into a search mode until it finds a blob. The blob data history not only helps to provide usable data during temporary outages from the proxy, it also provides a delay before the behavior will go into search mode. The search mode holds an internal state as well. If it has not been entered for longer than some pre-defined amount of time, it will reset its **spiral_starttime** variable. The search mode outputs a spiral trajectory in an effort to direct the Rover in a path that will make the blob visible if it is within blob-tracking range. To summarize, the **spiral_starttime** value and current time are used to calculate elapsed time in search mode and this value is used to calculate a trajectory vector direction component that given a constant vector magnitude component should result over time in an approximately spiral path. Instead of using time, distance from the starting point (radius of concentric circles along the spiral), could be used to travel a more exact path (Archimedean spiral), but this was not done because dead-reckoning was not available and GPS does not provide the necessary precision. During testing, the search mode was used several times by the Rover to point it in the right direction when it transitioned into blob-tracking mode pointing away from the CanSat.

sonarWander

This behavior was not used by the real Rover because the physical device (ultrasonic ranger array), was not ready in time. It was used in simulation (Stage), and provided good obstacle avoidance mostly for stationary objects either when the path planner was not in use, or for unforeseen obstacles not residing in a no-go zone. It operates by finding open areas (those containing relatively less obstacles than others), in the sensing area, and calculating a trajectory towards the biggest one.

The behavior implements some policies regarding its output: if there are two equally empty areas it will just pick one randomly instead of getting stuck. If all transducers are reporting a maximum reading (infinite) it outputs a zero trajectory because in the absence of obstacles it does not need to be active. If even one transducer is in an error state (no valid data available), the behavior also outputs a zero trajectory because it is not considered safe to make a decision about a good trajectory.

The behavior uses the following algorithm to find empty areas: each transducer points to an angle relative to the center-line of the Rover where 0 is pointing directly forward. Each transducer's reading will represent obstacles sensed in its corresponding zone (an arc of varying width, with our device about 15 degrees). Readings are treated in a discrete manner; above a certain threshold a reading will be considered “infinite” (no obstacle). A reading below that threshold will mark its zone “occupied”. There is no in-between.

Using these classifications, readings are analyzed to find the largest “open” region. As was stated before, if there are 2 or more equally large open areas, the algorithm will choose one at random. For the output vector, the direction will be the average of transducer angles for the open zone – falling in the center. The magnitude is directly proportional to the ratio of “open” zones to total zones; when all zones are open, magnitude will be full-speed (1), and when all zones are occupied the magnitude will be 0. This makes for interesting behavior when encountering obstacles. In the absence of obstacles the Rover will only be directed by other behaviors. If any one obstacle occupying the sensing area is detected, the Rover appears to speed up towards an open zone. As more obstacles are detected the Rover's speed quickly decreases. If the Rover drives right into a concave obstacle field it could get stuck as its speed approaches zero and no open zones can be found. Due to the environment the Rover is expected to operate in, this is unlikely. Testing in simulation demonstrated that this behavior worked well in sparse obstacle fields of stationary objects. If the Rover were in a very cluttered environment, it is possible this obstacle avoidance could fail, but then again, so would many other components (blob-tracking will fail if the blob is not visible). Another behavior could be added, such as avoidPast as one solution for the potential “getting-stuck” problem. Due to the nature of the task and time constraints, ultrasonic obstacle avoidance was given a lower priority on the Rover.

waypointTracker

This behavior uses data from a GPS proxy reading from a "gps beacon". With the Rover this comes from the radio signal sent by the CanSat and is processed to extract GPS data – UTM coordinates and altitude. **waypointTracker** takes this data, performs some filtering and applies inference rules to decide what the data state is and whether it can be used to calculate pre-conditions and input for other behaviors (although it has no knowledge of the other behaviors). This behavior is very specific to the nature of the task the controller is designed for. It runs as a state-machine, transitioning through modes as triggered by data events.

The defined data states are NONE, TENTATIVE, and GOOD. After it is initialized the behavior waits for any valid data from the proxy (NONE state – no data available). The first valid reading is stored and the machine enters TENTATIVE mode. Any subsequent readings will continue to be stored but data will not be made available to the main controller. In order to transition to the GOOD state, either of two events must occur: there must be a timeout, or the altitude of the gps beacon must fall below a pre-defined threshold. This is so that data will not be provided that is not usable; it is not possible for the Rover to reach a goal that is 1000 feet A.G.L., and the goal's 2-D coordinates may change substantially by the time it lands. Since the goal cannot move to a higher altitude, once in GOOD mode **waypointTracker** will remain in that state indefinitely, although it may issue warnings if the data becomes stale. It is also possible to stay in the NONE state indefinitely if no data is available at all. The TENTATIVE state will not last forever

because of the timeout. At some point, in the absence of fresh usable data the behavior will report its best estimate of the goal location. One improvement that could be made with the algorithm is to base the estimate on a more sophisticated prediction. Now the estimate just reports the last known data. Since the CanSat's trajectory is likely to follow a predictable trend, i.e.; down in the direction the wind carries it, a better estimate could be made.

aStarSearchMap

Given a start and goal, this behavior uses a map to calculate an “optimal” path that avoids marked “no-go” zones. It outputs either a solution set or an error if no solution was found. The solution sets are waypoints that are then relayed to the **waypointAttract** behavior. **aStarSearchMap** will encounter an error if it takes too long to calculate a solution and it times-out on its response, or if no-go zones make it impossible to travel a path between start and goal without traversing at least one of them, or if either start or goal fall outside of the map area. Since this behavior is called repeatedly throughout a mission, it is possible for no solution to be found on one call, then the next call to be successful depending on how start and goal have changed. The way **aStarSearchMap** is configured, the map itself (and no-go zones contained within it), cannot change during run-time, but this could be changed fairly easily to allow updates. One of the ideas would be to update no-go zones with other forms of obstacle detection (vision, etc..), but this remains to be implemented.

armPickup

This is not really a behavior in its own right but is included for consistency. It is implemented as a state machine. At start, the arm state is up, down, or unknown. If it is not up, the command is issued to lift it up. After a pause it is brought down. Then after another pause it is lifted back up (hopefully with payload in tow). At that point control passes back to the main controller.

3.2.2.2 Abstract Behaviors

The Abstract Behaviors were given less representative names – 1, 2, and 3. Their names are the order that would be followed during a complete mission where everything works perfectly. Given unexpected circumstances it is entirely possible to start at 1, then go to 2, then back to 1, then 2 again, and so forth. It is also possible to run partial missions; during testing only 3 can be run by itself, or 1 then 3, or just 2. In order to complete a full mission, 3 must be the final behavior that is active.

Here are some descriptions:

- 1 – use **waypointAttract**, **sonarWander**, **waypointTracker**, and **aStarSearchMap** to traverse a path bringing the Rover within a short range of the goal (payload), without hitting no-go zones or unforeseen obstacles. This is the first phase of the “long game”.

- 2 – use **waypointAttract**, **cameraAttract**, and **armPickup** to drive the Rover right up to the payload and pick it up. This is the “short game”.
- 3 – use **waypointAttract**, **sonarWander**, **waypointTracker**, and **aStarSearchMap** to traverse a path bringing the Rover within short range of a pre-determined flag waypoint without hitting no-go zones or unforeseen obstacles. This is very similar to behavior 1 and is the final phase of the “long game”.

3.2.2.3 Role of A* Planner

Of all the primitive and abstract behaviors in the controller, the most deliberative is the path planner. The path planner's specific task is to generate a list of consecutive waypoints traversing an “optimal” path between given start and goal waypoints without traveling through a given set of no-go zones (represented as marked cells).

The planner is based on the well-known A* algorithm used in many other optimal plan calculation tasks. A* worked well for the Rover's task and operating range; path calculation could be performed during run-time on a relatively slow processor without incurring any significant performance penalties. This was achieved in two ways: 1) a new path is computed at a low frequency, 2) the planner grid size (number of cells), is kept fairly small. The path planner was only invoked about every 20 seconds during a typical full test spanning about 2 miles roundtrip, and could have probably been called less often if optimization was deemed necessary. In order to use appropriate grid sizes and to assist in generation of usable map data, an “A* Map Tool” web application was developed and

deployed along with the controller. (Figure 3-2) This typically created grids with cell sizes of around 70 meters, a good balance of granularity and efficiency for the Rover's task; no-go zones are mostly a small number of large obstacles, while the size of the task space is relatively large (several miles).

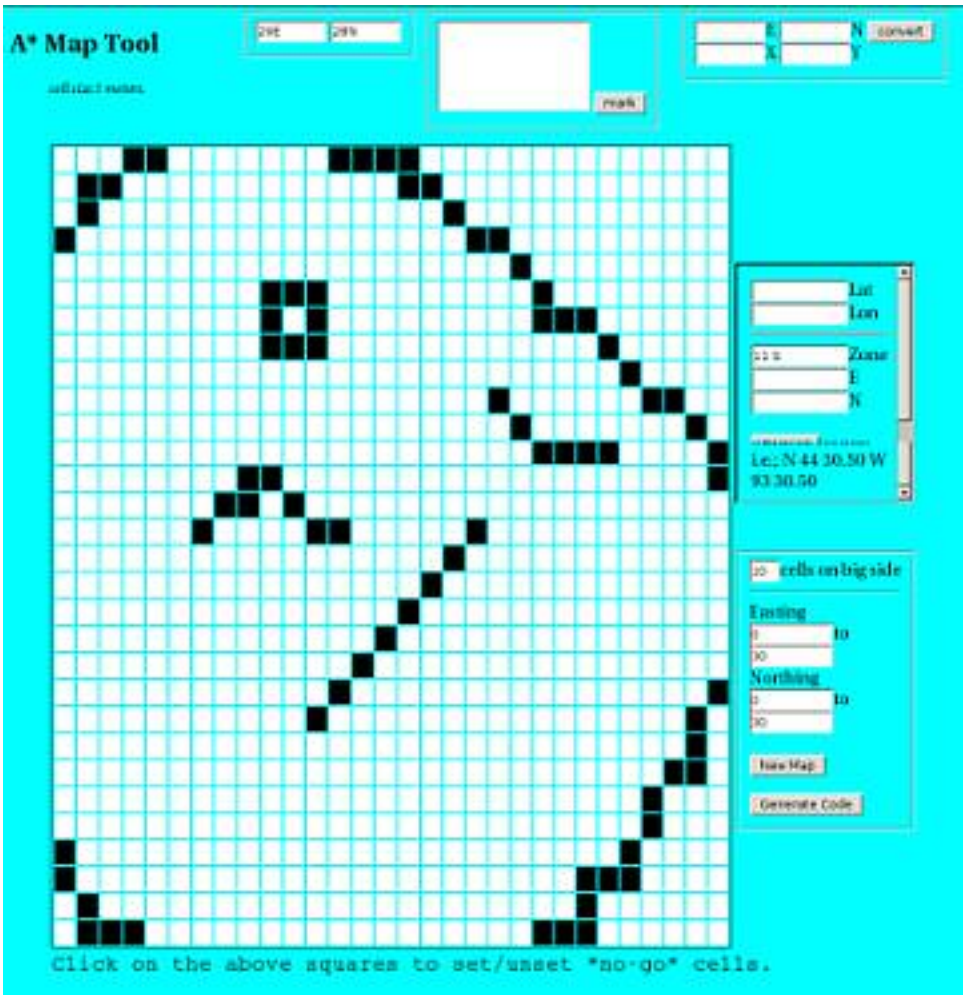

Figure 3-2 – A* Map Tool Used to Build Maps for Path Planner

3.2.2.4 Integration

The controller architecture could be considered a **hybrid** architecture because it has both reactive and deliberative components. What differentiates it is the structure in which these components are integrated. There is no subsumption or control hierarchy between reactive and deliberative components, they operate at the same level. What makes them different is the manner in which they perform their tasks, i.e.; the **waypointAttract** behavior maps sensor input and configuration requests directly to its output using a set of rules and formulas with no deliberation, while the **aStarSearchMap** behavior uses a symbolic representation of the real world (the map), and deliberates about its output (the recommended set of waypoints). The main controller just facilitates the flow of data to result in a final action for the robot. The waypoint list from **aStarSearchMap** is relayed to **waypointAttract** allowing it to generate a trajectory. This is merged with trajectories from the other active behaviors to form a final trajectory.

Some of the reactive components themselves blur the distinction between reactive and deliberative. This mostly involves the use of state machines and inference rules. Behaviors which hold no internal state are the best examples of purely reactive control. There are some however which may appear a bit more deliberative – for example **waypointTracker** transitions through a set of states based on data it receives about the waypoint (goal object) it is tracking, and a set of policies, in order to make a "decision" about the data. If no data has been received from the CanSat, it waits and reports that the waypoint is not valid. If data is received, it will wait until the data meets certain

requirements up to a timeout. When the timeout is exceeded it will just report whatever is available. In these ways, it acts as an “intelligent” filter for the incoming data. This filtering is its task and produces its output, similar to the output produced by the more purely reactive behaviors. The integration of primitive behaviors regardless of how deliberative or reactive, is performed by the specification of abstract behavior composition, data sources for preconditions, and primitive behavior weights.

3.2.3 Self-configuration

During initialization, a set of inference rules are applied to configure the controller for the mission it will be performing. The rules are applied to the set of requested tasks and to available resources. If certain resources are not available the rules will imply that the tasks which require these resources will not be performed even if they were requested; requests that are not possible will be ignored, or in certain cases an error will be reported and the mission will be aborted. Here is an example: say a Player configuration file is used that provides a gps device to track the gps beacon (CanSat), an arm, and a blobfinder device, but does not provide localization devices for the Rover (on-board gps and compass). In that case it will be implied that waypoint tracking will not be performed, and by process of elimination, only blob tracking and acquisition by the arm will be possible. The controller will then configure itself to perform a task involving only those phases. If there is also no blob tracking device available, only the arm will be used. Of course if there is also no arm, the controller will assume there is nothing to do and report that it has completed its mission. Conversely if a gps beacon, localization, and a blob tracker are available but there is no arm, then the Rover will configure itself to

complete all tasks except acquisition with the arm. All decisions that are made based on these rules will be logged as they are made. In this way if a mission is run and the Rover's behavior is different than expected it might be traced to decisions made based on the devices that seemed to be available.

The self-configuration feature may also become active during the mission when it is discovered that a resource is not available. Here is an example: normally if an A* planner is declared for the controller, it will be active during waypoint tracking phases of the mission, providing the next waypoint to be sent to the **waypointAttract** behavior. If the planner is at some point unable to provide a path for whatever reason, the rule is that the goal waypoint will be passed directly to **waypointAttract**, circumventing the planner and allowing the mission to continue in an appropriate manner.

Similar inference rules may also operate at other levels in the controller architecture, separate from the main controller. Here is an example: the **sonarWander** behavior is used during simulations run in Stage to provide obstacle avoidance in the absence of a planner or when avoiding unforeseen obstacles not found in no-go zones. Although Stage provides simulated ultrasonic ranger array data, this hardware device was not available for the Rover at the writing of this thesis. It was never tested in the real-world. When the controller initializes, it checks the Player configuration file and finds no sonar device. The controller code still declares a **sonarWander** behavior object. The behavior itself will notice that it is being passed a nil pointer to the proxy providing its input data, and always return zero for its output vector. There are other conditions in which the primitive

behaviors, upon non-fatal errors, will just return zero outputs. All of this is done in order to allow the Rover to recover gracefully from device failures during a mission.

Self-configuration was very useful during testing. The Rover could perform partial missions based on the resources it had available. Even before the arm was built it was possible to test the other phases of the mission and verify that the controller followed the right sequences without changing the controller code. When the Rover completed the blob-tracking phase, it would stop, wait a few seconds, then assume the CanSat had been acquired and continue with the next phase of the mission. When the arm was added, the controller just transitioned to using the arm with no modifications to the code.

3.2.4 Sequencing Through the World

The nature of the **Come-back Competition** task for the Rover is very sequential. The task has a well-defined succession of events. The Rover must travel to the CanSat in the beginning, then it must acquire it, and finally it must return the CanSat to a flag waypoint in the end. For the most part, these phases must be performed in that order. It would seem easy to hard-code that sequence into the controller, or perhaps read it from a file, or maybe use a sequencer to plan the transitions from phase-to-phase. This was not done for two reasons: hard-coding a sequence would make the controller more rigid and intolerant of unexpected events. If a sequencer were added, it would diminish the computational efficiency of the mostly reactive architecture while introducing further points of failure and complexity inherent in its deliberative process.

Instead, sequencing is implicitly performed by the set of preconditions controlling the activation of the abstract behaviors. This allows the controller to recover from unexpected events almost limitlessly, and frees its designer from having to consider all of these unforeseen circumstances. It is possible to write the abstract behaviors as follows – if condition A is true and condition B is false, then abstract behavior 1 is active. Otherwise if conditions A and C are true, then abstract behavior 2 is active. This makes the preconditions that much more important. Deciding what preconditions to use and inferences to make based on them is crucial in designing the controller successfully. Here is an example: probably the most important precondition for the Rover controller is **cansatAcquired**. If **cansatAcquired** is true, then only behavior 3 can be active since the only task remaining is to return to the flag. If **cansatAcquired** is false then either behavior 1 or 2 should be active. Whether this is due to the fact that the CanSat is yet to be found and acquired or that the CanSat was found, acquired, and dropped 10 times already is irrelevant to which behaviors should be active. Based on the set of precondition states, the right sequence will be implied and performed indefinitely (or until the batteries go dead).

3.2.5 Generalization of Controller

The Rover controller as a whole, that is, all of the software components used to control the Rover, is designed to perform a very specific task. The main controller structure itself however, is just an endless loop that uses output from agents to update preconditions, uses the preconditions to activate abstract behaviors, then forwards the resulting commands to the actuators. This structure is quite generic. It plays no role in the control

of the robot other than to act as a "facilitator" for the primitive and abstract behaviors to do their jobs. It would be possible to provide it with a different set of components, and use it as a controller for a completely different task. It is the particular collection of the components that results in a highly specialized controller. These components can be summarized as follows:

- resources:
    - agents - a library of primitive behaviors providing reactive or deliberative services such as monitoring/filtering of incoming sensor data for the purpose of setting preconditions, calculating actuator commands based on sensor data, or planning a path; tasks that are the foundations for completing an overall task
    - devices – sensors and actuators. Actuators provide mechanical capabilities, while sensors provide data about the environment. During development and testing, a sensor can be replaced by a file containing the data the sensor would provide. Stage is an active example of this. Actuators can also operate in a simulated world, providing an observer information about how the actuators might interact with the real world
- abstract behaviors – compositions of agents and the weights applied to their outputs, that result in further abstraction of functionality than could be provided by any one agent alone. a specification of the preconditions that activate them
- preconditions – a specification of which agents provide them and how they are calculated.

- inference rules – based on available resources: agents, tasks, and devices; the controller self-configuration decisions to be made and how to make them

With different components, it would be possible to use the main controller that controls the Rover to perform a completely different sequential task. An example of this might be to control a robot that washes a car. The components would include the following:

- resources:
    - water sprayer
    - arm with brush
    - soap dispenser
    - camera
    - obstacle detection (sonar or laser)
    - motorized vehicle
    - localization
- agents:
    - sprayWaterOnCar
    - driveToLocation
    - avoidCar (do not collide with it)
    - brushCar
    - monitorSurface - monitor car's surface condition (i.e.; dry, wet, dirty, clean)
- abstract behaviors – these could be state-less or hold an internal state. They would compose agents into resultant behaviors corresponding to phases of the overall

task. For example, an abstract behavior could combine driveToLocation, avoidCar, sprayWaterOnCar, and monitorSurface to rinse all soap off the car after brushing it. Another abstract behavior could combine driveToLocation, avoidCar, brushCar, and monitorSurface to brush the car with soap.

- preconditions:
    - car_is_wet
    - car_is_clean
    - car_covered_with_soap
    - car_brushed_down
- inference rules: these would apply in the presence or absence of resources, and would be useful during development. for example, if no camera were available then one inference could be that there is nothing to do because there is no way to tell whether the task has been completed, so the controller would just exit. Another inference might be that if no brush is available then brushing will not be performed and car_brushed_down is set to true overriding monitorSurface. That way it would be possible to test wetting down and rinsing by themselves without changing the controller

This example demonstrates that the same generic controller architecture could be used to develop and perform execution of completely unrelated tasks by changing the underlying components in a consistent manner.

3.2.6 Robustness and Reusability

The highly modular design of the controller software made it possible to test components individually, and add, remove or exchange them for others while keeping the same overall structure intact. This was indispensable during development, as having to test an entire mission every time would have made the task many times more cumbersome. Also, not all hardware devices could be expected to be operational at all times. Many were not ready during development, or broken, some were exchanged for others. Player/Stage facilitated and influenced the modularity of the design by decomposing sensing and actuating into layers – hardware device to operating system driver to Player driver to Proxy Interface. From there further layers in the controller used proxies in the agents, which were composed in abstract behaviors, which were all fused by the main controller. This all makes it possible to change any one component – hardware, driver or agent, etc.., without having to modify the rest of the system.

Resilience to unexpected events is built into the controller architecture. Not only does high modularity and low cohesion accomplish this, it is also built into the policies that guide abstract and primitive behaviors. The filtering of data performed by the **cameraAttract**, **sonarWander**, and **waypointTracker** hides the low-level errors in sensor data from higher-levels in the architecture, allowing them to operate with more stable information.

Resilience to unexpected events is also a product of “sequencing through the world” by using pre-conditions. It is not necessary to hard-code a sequential procedure or require

the controller to deliberate about what to do next. This frees the controller from processing overhead and the developer from having to account for all the possible states, whose numbers increase as the task becomes more complex. It makes it possible for the controller to iterate through phases of the implied sequence in different orders until the task is completed, indefinitely. The pre-conditions are based on direct sensor data instead of abstract representations of the world, reducing the possibility that decisions will be made based on a false representation. Careful definition of the pre-conditions and the abstract behaviors they control is a critical task, but once complete, sequencing through the world allows the developer to focus on fine-tuning the individual components, resulting in a more flexible and robust overall design. Finally, since the abstract behaviors fuse outputs from more than one source into a final output, they prevent any one malfunctioning primitive behavior from completely controlling the robot. If one primitive behavior is not leading the robot to perform its task, another might do it.

Chapter 4 Results from Final Testing at White Lake, NV

4.1 Methods

Testing was performed on individual components throughout the life of the project, in both simulation and the physical environment. This section will focus on the testing of the entire unit that was performed in later stages of development in a physical environment similar to that where the ARLISS **Come-Back Competition** is held. Over a period of about 3 weeks a total of 11 tests were performed at White Lake, a dry lake bed

near Cold Springs, Nevada. This environment is very similar to the Black Rock Desert where ARLISS is held, on a smaller scale. Actually, many more than 11 tests were performed during that time period, and especially during the months before final testing, adding up to a few dozen hours. These are not included here because they were not documented. They were considered to be just part of the design process. As it turns out, even the final tests ended up affecting the later stages of development since they exposed bugs that were not detected without very thorough testing. These bugs were corrected and will be discussed in more detail a bit later. The following types of tests were performed:

- Long-game only (total 2) - the Rover had to drive out to the CanSat and back to a flag location based only on gps/compass. It was not required to blob-track or use the arm. These tests included A* path planning.
- Short-game only (total 3) – the Rover was placed within blob-tracking range of the colored parachute and CanSat. In some of these it was required to drive to it and pick it up with the arm, in the others it only did blob-tracking.
- Complete Mission (total 6) – the Rover was required to perform a task as close as possible to the real Come-Back Competition.

For the most part, the physical components that would be found in an ARLISS were used during testing – the robot and all its hardware, an CanSat with parachute on the ground, radio transmission from a CanSat, wind, sun, and the terrain of a dry lake bed. The only exception was that it was not possible or necessary, to launch a real rocket deploying a CanSat as a payload in order to perform the tests. To simulate the circumstances of a real

launch, a small tower was constructed next to the CanSat/parachute location on the ground and a second CanSat was placed on top of the tower that would do radio transmission as the “gps beacon” (Figure 4-1). The tower was necessary because on the flat lake bed radio signals will not travel much further than a few hundred meters without elevating the radios or enhancing their hardware. The goal was to make the test as close as possible to the actual competition, and this would – a CanSat deployed from a real rocket would fall from above and provide very good radio reception as it fell from ~10,000 foot A.G.L.

Figure 4-1 – Radio Tower Assembly Used to Mimic a Falling CanSat

Short-game tests were simple and straight-forward. A CanSat and colored parachute were placed on the ground within blob-tracking range of the Rover. The Rover was configured to perform the short-game task, sometimes starting in a position where the parachute (color blob), was in its field-of-view, other times with the Rover facing in the opposite direction so it had to search first. When the Rover reached the perceived blob it would attempt to pick it up with the arm, and the test would finish.

Figure 4-2 – Cone Marking Edge of a “No-Go” Zone

Long-game tests involved a little more work; a tower was set up, CanSat / parachute placed on the ground next to it, and the Rover placed about ½ mile away. For tests including path planning, a no-go zone was defined arbitrarily by acquiring GPS

coordinates for its endpoints, entering these in the A* Map Tool, then generating a map file. Small orange traffic cones were placed to mark the no-go zone (Figure 4-2). The cones were not used by the Rover, but allowed human observers to detect whether the Rover had planned a path around the no-go zone or not. The no-go zone was a straight line about 0.2 miles long, and perpendicular to the line between the Rover and the CanSat. The no-go zone roughly bisected that direct path between Rover and CanSat, insuring that if the Rover did not perform path planning, it would noticeably travel right through it.

4.2 Data and Discussion

The results show that separate short and long game tests had the most consistent success. Of 3 short game tests, all 3 were successful. Both long-game tests were also successful. It should be noted that these results do not tell the entire story. My observation was that of those 2 modes of operation, the long-game was much more reliable. Out of probably a dozen informal long-game tests that were performed, the Rover never failed to navigate waypoints to the CanSat, with or without A* path planning. Success during the short-game was more elusive. All by itself, short-game test results were consistent; the Rover would visually search its surroundings for the parachute color blob, find it, and if it had the arm available pick it up. This was because the camera settings were calibrated right before the test and lighting conditions had no time to change. In addition, if the arm was being used it had no time for the wind to blow it around. Problems occurring during the short-game were exposed with full tests.

Full tests were more difficult because 2 of the main challenges affecting successful completion of the challenge had a chance to take effect. The first involves the changing light conditions found in the outdoor operating environment; although the blob-tracking is an improvement over the previous generation of the Rover, it is the most fickle component of the controller. It must be carefully calibrated and can fail for various reasons. If light is too directional (early or late in the day), the color sampling done by the OpenCV blobfinder driver may fail because different light intensities will reach the color sample and the parachute causing them to appear as very different shades. This then causes blob tracking to either completely miss the parachute, or worse, to "see" it where it is not. If a shadow is cast on one item (sample or parachute), but not the other, the problem will almost surely occur (Figure 4-3).

Figure 4-3 – Failed Complete Test Due to Blob-tracking Error

The other problem involves the arm and what happens to it if conditions are too windy. The arm, being in very much of a prototype stage as of the writing of this thesis, does not hold its position if a force is applied in its range of motion. The only source for this type of force during a mission would be the wind. Winds during a typical summer day on a dry lake bed in Nevada tend to be calm early after sunrise and get progressively stronger later in the afternoon. By sunset it is not unusual for winds to be gusting 30 or 40 miles-per-hour even if there is no storm passing. The winds can kick up dust storms and become strong enough to make outdoor activities grind to a halt as was evidenced during the 2005 ARLISS. When winds get too strong the Rover's arm gets pushed down, into the field-of-view of the camera. This can make blob-tracking impossible or prevent the arm from having enough strength to lift the CanSat.

Of the 6 full tests that were performed, half were completely successful. 2 failed because of problems relating to blob-tracking. The arm was in a position where it cast a shadow on the color sample, causing its color to appear very different from that of the parachute on the ground. In the first, the Rover was driving too quickly when it drove up to the parachute, lost sight of it, then ran over it when it started its search routine. This was a bug that was fixed. Since the CanSat was underneath the parachute, the Rover got caught up in the parachute and stalled, requiring human intervention. In the second failed full test, the Rover went from short-game to long-game (started blob-tracking), and immediately spotted a suitable blob on the distant range of hills. Since the blob also fit the criteria for a “reached” blob, it deployed the arm, assumed the CanSat had been

retrieved, then continued onto the next phase back to the flag without the CanSat. This situation may have possibly been remedied by having a more definitive way to know whether the CanSat had actually been retrieved.

The third failed full test was due to a hardware problem (the compass was accidentally reset). The box containing the circuit board for the compass has a reset button on the outside. If this is accidentally pressed while the box is powered on, the compass will reset and begin reporting erroneous readings. That is exactly what happened; the button was accidentally pressed before the test began without our knowledge, and the Rover began the test with the bad data. Consequently, it was impossible for it to drive in the right direction to perform GPS waypoint navigation and the test failed right from the beginning. The cause for the failure was not discovered until later that day.

3 of 6 full tests were completely successful. The Rover completed all the tasks that would be involved in a real Comeback-Competition, returning with the CanSat to the flag with no human assistance. Each of these tests spanned a round-trip distance of over 2 miles, and took approximately 20 minutes to complete. The Rover was able to plan a path around the predefined “no-go” zone, used the radio data modem to successfully track the CanSat's location, transitioned through abstract behaviors using preconditions to the short-game, then deployed the arm to grab the CanSat and place it in the basket. Since abstract behavior “3” was active, it drove back to the flag location, and stopped within about 10 meters by itself. Logs showed that the controller exited normally.

Chapter 5 Future Work

As was demonstrated by the tests, there is room for improvement with the current design of the Rover. The most immediately obvious areas are with the blob-tracking component and with the arm. The arm is mostly a mechanical issue. As was mentioned earlier, the arm is in very much of a prototype stage. It was constructed during the two weeks before final testing, and although it is a decent design, it barely does the job in its present state. The arm would work better if it could hold itself in place, perhaps with a servo, and had greater torque so it could reliably pick up the parachute and CanSat even in strong winds. This would also keep it from interfering with the camera and disrupting blob-tracking.

The blob-tracking is more of a challenging area for improvement. At present, the blob-tracking Player driver that was written specifically for the Rover, uses a physical color sample as an adaptive mechanism to address changing light conditions. This mechanism is fairly successful in uniform lighting conditions (indoors or when the Sun is overhead), but is subject to failure in more directional lighting or when affected by the arm. There are probably better computer vision techniques that could be applied to perform this task, but they are yet to be designed and implemented on the Rover.

Another area for future work is with obstacle detection and avoidance. Although ultrasonic range-finding was used successfully during simulation (in Stage), the real sensor was not completed on the Rover and was not tested. The primitive behavior **sonarWander** was demonstrated to be effective in simulation. It could be used in the real

world when the hardware device is ready. In addition, the obstacle detection could be integrated with the path planner by enhancing the planner to allow loading of new maps or modifications to the existing map during run-time. Unforeseen obstacles that are detected on-the-fly could be added to the map by a new primitive behavior – a mapping behavior. This would allow the Rover to dynamically plan paths around obstacles as they are detected, enabling it to operate in more cluttered environments.

RFID, or **Radio-Frequency Identification**, is a type of sensor that is gaining in use in many applications from robotics to product tracking. With the Rover, there are plans to use it to allow the controller to identify the CanSat when it is within close range. As of the writing of this thesis, there are Player drivers available to facilitate the use of one of these devices, and the behavior to utilize them would be very straight-forward to write. This behavior could monitor whether the CanSat has been detected, providing more reliable data to determine whether the CanSat is still acquired by the Rover. One difficulty might be in fitting the space-restricted CanSat with the right transmitter.

Chapter 6 Conclusion

Development of the current Rover for the Nevada Student Satellite Program has resulted in important successes, especially with the limited resources that were available. We developed an autonomous robot that is capable of executing a fairly sophisticated task in an unstructured outdoor environment, at a much lower cost than robots with similar capabilities. The Rover's controller architecture builds on previous work in Behavior-based Control, combining best features of both reactive and deliberative components with

minimal hybridization. Finally, the modularity of both hardware and software components leaves room for enhancements that can be made incrementally, without disrupting the overall controller structure.

Bibliography

[Arli07] http://www.arliss.org/ Accessed August 8, 2007

[Neva07] http://www.unr.edu/NevadaSat/ Accessed August 8, 2007

[Play07] http://playerstage.sourceforge.net/ Accessed August 8, 2007

[Arki98] Arkin, R. C. (1998). Behavior-Based Robotics. MIT Press, CA.

[BrBr06] Braid, D.; Broggi, A.; Schmiedel, G., "The TerraMax Autonomous Vehicle concludes the 2005 DARPA Grand Challenge," Intelligent Vehicles Symposium, 2006 IEEE , vol., no., pp. 534-539, 13-15 June 2006

[Broo91] Rodney A. Brooks, Intelligence Without Reason, Massachusetts Institute of Technology, Cambridge, MA, 1991

[LaWa07] Jeff C. LaCombe, Eric L. Wang, Monica Nicolescu, Pablo Rivera, B. Poe "Design Experiences with a Student Satellite Program", in Proceedings, the American Society for Engineering Education, Pacific Southwest Section Conference, Reno, NV, USA, April 12-13, 2007.

[Mata92] Maja J Mataric´, "Integration of Representation Into Goal-Driven Behavior-Based Robots", in IEEE Transactions on Robotics and Automation, 8(3), Jun 1992, 304-312.

[Mata98] Maja J Matarić, "Behavior-Based Robotics as a Tool for Synthesis of Artificial Behavior and Analysis of Natural Behavior", Trends in Cognitive Science, 2(3), Mar 1998, 82-87.

[NiMa02] Monica N. Nicolescu and Maja J. Matarić, “A Hierarchical Architecture for Behavior-Based Robots”, AAMAS’02, July 15-19, 2002, Bologna, Italy.

[Rayb07] "Field Demonstrated Autonomous Robot Control for CanSat Rocket Payload Retrieval", Rayburn, Christian. Masters Thesis, University of Nevada, Reno, 2005.

[ReMa06] Redmill, K.A.; Martin, J.I.; Ozguiner, U., "Sensing and Sensor Fusion for the 2005 Desert Buckeyes DARPA Grand Challenge Offroad Autonomous Vehicle," Intelligent Vehicles Symposium, 2006 IEEE , vol., no., pp. 528-533, 13-15 June 2006

[Rose97] J. K. Rosenblatt. DAMN: A Distributed Architecture for Mobile Navigation. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1997.

[Stee90] Steels, L. 1990. Towards a theory of emergent functionality. In Proceedings of the First International Conference on Simulation of Adaptive Behavior on From Animals To Animats (Paris, France). J. Meyer and S. W. Wilson, Eds. MIT Press, Cambridge, MA, 451-461.