

BotKit: The Robot Construction Kit

by

Edwin W. Foo

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Electrical Engineering and Computer Science

at the

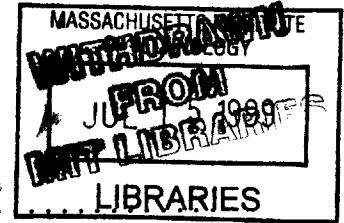
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1999

© 1999 Edwin W. Foo. All rights reserved.

ENG

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.



Author
Department of Electrical Engineering and Computer Science
May 21, 1999

Certified by
James E. Hicks
Research Staff: Compaq Cambridge Research Laboratory
Thesis Supervisor

Certified by
John Chapin
Assistant Professor: Laboratory for Computer Science, MIT
Thesis Supervisor

Accepted by
Arthur C. Smith
Chairman, Department Committee on Graduate Students

BotKit: The Robot Construction Kit

by

Edwin W. Foo

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 1999 in Partial Fulfillment of the Requirements for the Degree
of Master of Engineering in Electrical Engineering and Computer Science

Abstract

Embedded systems exhibit reactive behaviors that can rapidly respond to stimuli, yet must keep a long-term goal in sight. These requirements make software development for such systems difficult because of the conflicting short-term and long-term goals. The subsumption architecture attempts to solve this issue by presenting a system in terms of a network of horizontally layered behaviors implemented using Augmented Finite State Machines (AFSMs). The subsumption architecture works well but has not found its way into widespread use; one possible reason is the mindshift required to express behaviors in terms of a state machine. Another barrier is the traditional implementation of subsumption architectures in hardware or specialized programming languages. This thesis describes an attempt to address both issues using a subsumption architecture implemented in Java as well as an encapsulation system designed to allow fragments of robot behaviors coded using traditional methods that are to be embedded in AFSMs. The approach allows a bottom-up incremental approach to system implementation while encouraging an eventual reimplemention from the top down. This was also introduced to contestants in the 1999 6.270 Autonomous Robot Design Competition with some degree of success.

Thesis Supervisor: James E. Hicks

Title: Research Staff: Compaq Cambridge Research Laboratory

Thesis Supervisor: John Chapin

Title: Assistant Professor: Laboratory for Computer Science, MIT

Acknowledgments

Jamey Hicks was my primary advisor and mentor during my work on BotKit and the RoboSkiff controller at the Compaq Cambridge Research Laboratory (CRL) in Cambridge, Massachusetts. He greatly influenced the design and implementation of BotKit, and was project lead for the Skiff controller project at CRL. John Chapin at the MIT Laboratory for Computer Science was my on-campus thesis advisor and also provided guidance in the writing process. CRL supported my research through a graduate fellowship in the MIT VI-A program.

The rest of the engineering and research team at CRL who worked on RoboSkiff also deserves a word of thanks. Without these people, Skiff would have never become gone from a sketch on a piece of paper to a working board in three months. In no particular order, the RoboSkiff team is: Frank Bomba, Geoff Bomba (Frank's son), Ed Chang, Jack Costanza (MIT AI Lab), Donald Denning, Bob Iannucci (Laboratory Director), Chris Joerg, Jason Lee, Dave Panariti, and Mike Schnexnaydre.

I would also like to acknowledge the 1999 6.270 organizers for their efforts in making the 1999 contest the best one yet. They are: Mike Allen, Rob Blau, Andy Chang, Adrian Danieli, Grant Emery, Anthony Hui, Steve Paik, Gong Ke Shen, and Yonah Schmeidler. Charatpong Chotigavanich, Bruce Po, and Joyance Meechai also deserve special mention for going beyond the call of duty as 6.270 Teaching Assistants and helping with the boards during the contest. All the rest of 6.270 Staff also deserve mention; without them, the contest would never have happened.

The 1999 6.270 contestants deserve special mention for being willing to put up with prototype hardware and software and constructing working robots despite the odds. Everyone should be proud of you for pulling that off.

Last, but certainly not least, I give thanks to God for giving me the energy and peace of mind I needed to finish. Many of my friends at church and fellowship also gave a lot of support and encouragement when it seemed that nothing was working.

This thesis is dedicated to Irene Foo. Thanks for everything, Mom. I hope you get well soon.

Contents

1	Introduction	10
1.1	Embedded Systems	10
1.1.1	Introduction	11
1.1.2	Development Cycles	11
1.1.3	Real-Time Response/Reactive Control	12
1.1.4	Code Reliability	13
1.2	Robotics and Embedded Software	13
1.3	Motivation: The MIT 6.270 Contest	14
1.3.1	Author's involvement	15
1.3.2	Original 6.270 Controller History	16
1.3.3	Compaq CRL Involvement	18
1.3.4	Development Software	18
1.3.5	Final Software Specifications	21
1.4	Robot Controller Hardware/OS Requirements	22
1.4.1	Hardware	22
1.4.2	OS and Support Software	25
2	Models of Computation	27
2.1	The Line Follower	28
2.2	Procedural Models	29
2.2.1	Implementing the Line Follower	30
2.2.2	Analysis	30
2.3	Multithreaded Procedural Code	32

2.4	Finite State Machines	34
2.4.1	Advantages	35
2.4.2	Disadvantages	36
2.5	Multiple Concurrent FSMs	37
3	The Subsumption Model	39
3.1	Introduction	39
3.2	Advantages	40
3.3	Implementation	43
3.3.1	AFSMs	43
3.3.2	Alternatives to AFSMs	43
3.4	AFSM Disadvantages	44
3.4.1	Learning Curves	44
3.4.2	Small Pieces	44
3.5	AFSM Wrappers for Procedures	45
3.6	Forming BotKit	46
4	Implementation	47
4.1	Why Java	47
4.1.1	Drawbacks of C	47
4.1.2	Portability	48
4.1.3	Stronger Typing and Garbage Collection	48
4.1.4	Threads	50
4.1.5	Procedural Code	51
4.2	BotKit	51
4.2.1	Overview	51
4.2.2	Example Use	57
4.2.3	Line Follower + Distance Measurer	58
4.3	Implementation Notes	59
4.4	Time Frame	60

5	Results and Analysis	67
5.1	NetBSD/Kaffe combination	68
5.2	Java	68
5.3	BotKit	69
5.4	Future Work	71
5.4.1	FSM description tools/languages	71
5.4.2	FSM Verification	72
5.4.3	NetBSD/JVM work	72
5.5	Conclusion	73

List of Figures

1-1	A photo from the 1999 6.270 competition	15
1-2	The Old 6.270 Controller	17
1-3	An example Interactive-C session	20
1-4	The Personal Server motherboard (3.6 inches by 5.4 inches)	23
1-5	The RoboDC daughtercard	24
1-6	The assembled RobotController	25
2-1	A simple diagram of the line-following robot.	29
2-2	The simple line-follower algorithm implemented in Interactive-C.	31
2-3	A simple finite state machine. This diagram comes from Girault, Lee, and Lee's paper on hierarchical FSMs[13]	35
2-4	Some of the possible states the sensors can be in for the line follower. Note that two of the states in the figure correspond to odd situations that would be hard to catch otherwise if one were to just start writing the code without thinking the problem through first.	36
2-5	An example FSM network	38
3-1	The subsumption model.	40
3-2	The line-follower implementation using the subsumption model.	41
4-1	The AFSMState class.	52
4-2	The AFSM class.	54
4-3	The AFSMRegister class.	61
4-4	The AFSMNetwork class definition	62

4-5	The line follower as an AFSM	63
4-6	The line follower as a procedure encapsulated in an AFSM	64
4-7	Distance Measurer AFSM Implementation	65
4-8	Example robot combining Line Follower and Distance Measurer behaviors	66

Chapter 1

Introduction

This thesis describes an effort to make embedded system software development easier and more intuitive called BotKit. BotKit was developed for use in the MIT 6.270 Autonomous Robot Design Competition in conjunction with a new controller hardware platform called the Robot Controller.

Chapter 1 gives a general background on the 6.270 contest and the challenges that are commonly found in software development for embedded systems, particularly the robots used in the contest. Chapter 2 describes the different possible software models and introduces finite state machines as alternatives, where Chapter 3 goes on to introduce the subsumption architecture and how it can help model embedded systems. Chapter 4 goes over the implementation of BotKit and the Robot Controller 6.270 controller, and Chapter 5 presents results from deployment in the 1999 6.270 contest, and identifies areas for future work.

1.1 Embedded Systems

This section describes the basic ideas and challenges in designing and implementing an embedded system.

1.1.1 Introduction

An embedded system can be said to be a combination of hardware and software that forms a larger system and which is expected to function without human intervention. A typical embedded system consists of a single-board computer with software in ROM or Flash memory. This system starts running some special purpose application program as soon as it is turned on and does not stop until it is turned off.

Examples of embedded systems can be found in many home and industrial applications. Most control processes in industrial settings are run using embedded platforms and special purpose hardware/software. Automobiles also have a number of embedded processors controlling such things as engine ignition, fuel injection, anti-lock brakes, etc. Even children's toys now have small inexpensive embedded computers in them to enhance interactivity and flexibility.

These systems also tend to have unorthodox input/output ports, at least by desktop standards. Many of them do not have any of the normal peripherals such as a keyboard, monitor, mass storage, or user interface software. Instead, they interact with sensors and transducers that interface directly to the physical environment around them. Some examples are potentiometers, temperature sensors, etc. Often, the only recognizable connection to the desktop world is a serial interface used for configuration and reprogramming.

1.1.2 Development Cycles

Developing code for an embedded system is quite different from the process used for a desktop machine. Unlike a desktop machine, embedded controllers are normally incapable of running the compilers and development environments on the systems themselves. For example, a common embedded processor, the Intel 8051, has only 2KB of memory in it by default. A C compiler would be hard pressed to fit.

Therefore, most embedded development is done on other systems, normally desktops or workstations. These "host PCs" run cross-compilers that generate code for the embedded architecture, which is then downloaded to the target system via a serial

link and burned into ROM. Debugging facilities tend to be limited because there are typically not enough resources to load both the code and a debugging environment at the same time. The development cycle is also a bit longer due to the time needed to download a ROM image to Flash memory or transfer programs over a serial line before execution. As embedded systems start to come equipped with more onboard memory, co-resident debug stubs and monitors are more easily integrated into the system, but there will always exist a case for extreme space optimization to reduce cost that precludes use of onboard debuggers.

1.1.3 Real-Time Response/Reactive Control

Embedded system software has different constraints than desktop software. Some of these constraints make embedded code somewhat harder to design and implement. Since embedded systems are meant to operate in the background, users therefore expect them to simply work without having to stop and consciously interact with the system. Even when human interaction with an embedded system is required, instantaneous or near-real-time feedback is expected, not just to the user but in the system's response to the physical environment.

For example, a building thermostat has to continually poll sensors from various locations in an attempt to make decisions about how to control the heating and cooling systems to maintain a constant temperature. In addition to this long-range planning, the system has to respond to temperature-change inputs from users and immediately replan its heating/cooling strategy. Furthermore, the heating/cooling systems and the building are analog in nature and so do not ramp up and down instantaneously, so there is a feedback delay for changes the thermostat controller makes to the outputs. Extrapolate this type of situation to that of an aircraft engine controller that has to deal with turbines rotating at tens of thousands of rpm, and the response-time requirements become much more stringent.

1.1.4 Code Reliability

Embedded code also has to be extremely robust; this is not to say that desktop software can get away with extremely bad code, but the consequences of failures in embedded software are often higher. If an elevator gets confused and sticks between two floors, people will get annoyed. But if an airplane crashes because its engine controls get wedged in an unforeseen state, that is simply unacceptable. Many embedded systems operate under conditions in which failure is not an option, both literally and figuratively.

This emphasis on code correctness and robustness in the face of unfamiliar situations necessitates a very conservative approach in code development. Embedded systems in critical applications often are required by law to be certified line by line. Even for simpler systems, the number of potential inputs and outputs are very large due to their direct interface to the outside world. Something as simple as a VCR has to deal with sudden inputs from users like repeated presses of the Pause/Play buttons, ejecting a tape while playing, etc. Simply discovering all of the possible states in such a system is very difficult.

1.2 Robotics and Embedded Software

Robots are perhaps the most obvious application of embedded systems. The robotics field also presents embedded systems with some of its hardest problems. A completely autonomous robot is faced with real-time response requirements from multiple sensors, necessitating quick decision paths[11]. However, a higher-level thought process also has to guide overall robot behavior and account for momentary changes in state due to sensor readings. This makes software development for robots a particularly difficult process.

However, robot control software is capable of demonstrating many lessons about software engineering and design in a real-world environment. People who have been through this process usually have a healthy respect for what it takes to get code to survive in a demanding environment. At the Massachusetts Institute of Technology,

one class has attempted to introduce successive generations of students to the issues connected with robot control systems and embedded software development. This class is the MIT Autonomous Robot Design Competition, or simply 6.270¹.

1.3 Motivation: The MIT 6.270 Contest

6.270 is a month-long class in which students design and build a robot that will compete in a competition at the end of the month. The goal is for the students to design and build a machine out of LEGO² that can navigate the playing field, manipulate game objects, and interact (or avoid interacting as the case may be) with other opponents. 6.270 robots are completely autonomous and so must be programmed to operate on their own after a round starts; a small embedded controller and sensors are mounted to each robot to allow autonomous control. The kit is the same for each team; the amount of LEGO and sensors per robot is held to a constant maximum to encourage creativity and level the playing field.

The goal of 6.270 is to teach students about robotic design by giving them the hardware, software, and information they need to design, build, and debug their own robot. The class is very much a hands-on experience; students generally learn everything they need to know from hacking on their robots and working with each other. It is also the largest student-run event at MIT and attracts extremely large audiences every January when the contest is held. The class has also spawned many other classes and robotics organizations at other universities and schools around the world.

It is important to note that 6.270's mission is first and foremost an educational one. Besides providing contestants with a "real world" engineering problem and a limited time frame and budget, the class' environment is unique in its mix of theory and real-life practice placed in the context of a contest. Students also tend to find the course extremely engaging despite its high time commitment and arguably learn

¹Everything at MIT is commonly referred to by a number for brevity, including classes and buildings.

²LEGO is a registered trademark of the LEGO Group.

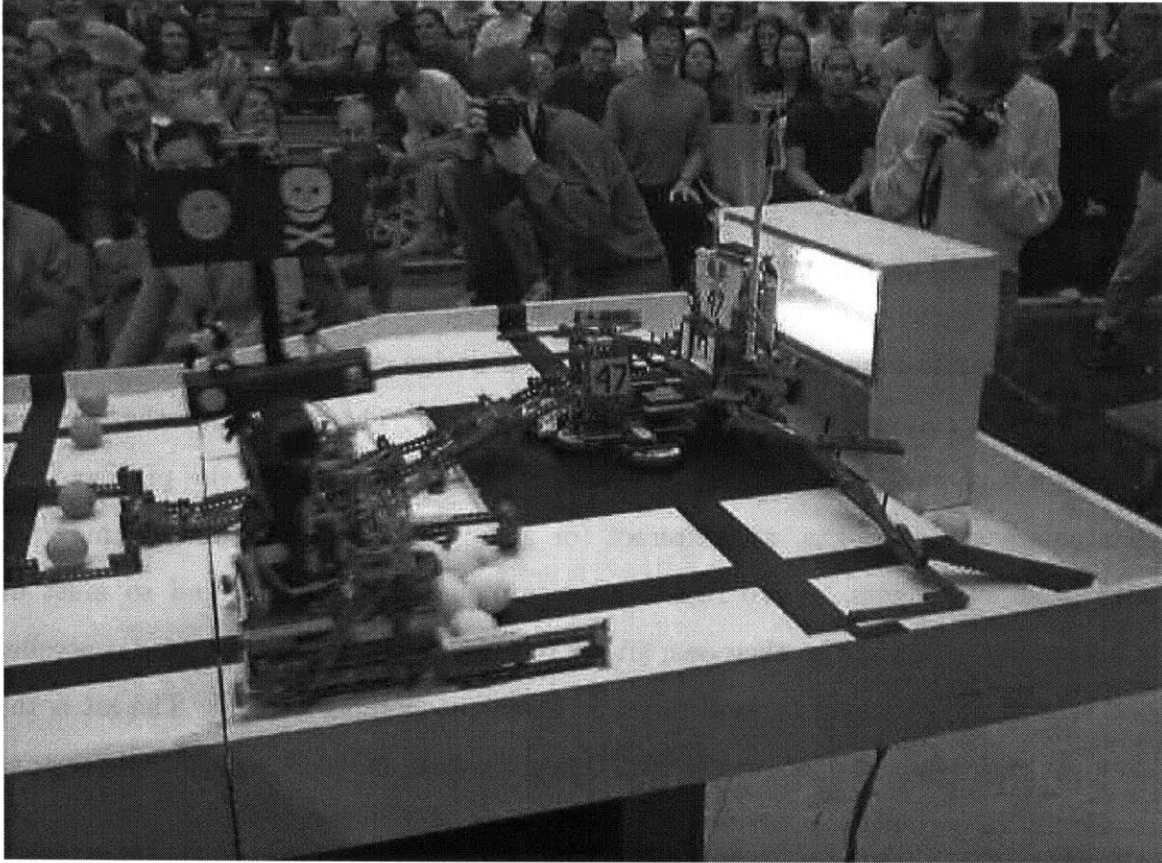


Figure 1-1: A photo from the 1999 6.270 competition

more about engineering than they would in a comparable classroom setting. It is not uncommon to see students staying up through all hours of the night working on their robots even though the class carries a negligible amount of academic credit. This is made more compelling when one considers that 6.270 is held during MIT's Independent Activities Period, which is a time generally reserved for fun and relaxing activities between the fall and spring semesters.

1.3.1 Author's involvement

I first competed in the 6.270 contest in January 1995 of my freshman year at MIT. My robot performed respectably, but a combination of hardware and software flaws caused it to lose and be eliminated from the competition. Most notably, I made a crucial mistake and assumed that a path in the code would always be followed before

a certain turn. Changes in the light level falling on the table code this code path to never execute during competition, so the robot never made the turn. If there had been a timeout or some other way to skip this step and continue with the rest of the routine, my robot would have probably done okay, but I failed to anticipate the possibility of not making the turn.

My experience writing the code for my team's robot left an impression on me, and the following year I convinced myself that I really wanted to learn how to design robust computer systems. I ended up volunteering again as a Teaching Assistant for 6.270 in 1997, and I have been a Contest Organizer for 6.270 from 1998 to the present.

In my capacity as a Contest Organizer, I have worked with the other Organizers on all aspects of the contest, ranging from running the entrance lottery to giving lectures, building the contest table, and speaking on contest night. However, my main interest has always been in the teaching and technology aspects of the contest. 6.270 provides students with an opportunity to interact with hardware and software in ways that to date have not been available in official MIT courses. The technology connected with the contest also has several characteristics that are not visible in other courses, like the Interactive C environment, direct control motors and actuators from a high-level programming language, and software interfaces to sophisticated sensors. The programming problems inherent in writing code for robots also present a hard educational problem in addition to a technical one.

1.3.2 Original 6.270 Controller History

The 6.270 contest first started as a completely simulated competition; robots faced off against each other on a virtual battleground. It was only in 1990 when the contest first featured fully autonomous robots running without any external help from computers or power sources[19].

The first 6.270 controller board (shown in Figure 1-2) consisted of a Motorola 68HC11 microcontroller with 32 kilobytes of SRAM and assorted analog-to-digital converter chips, motor drivers, and input/output ports. It was programmed using assembly language; no doubt giving some contestants problems, to say the least. In

fact, conversations with some contestants from that era have revealed that the reason some of the robots that year looked and acted the same was that one team got their assembly code working, and many teams just copied them and were too afraid to try modifying it. Fortunately, the following year saw the introduction of a small interpreter that allowed robots to be programmed using a subset of the C programming language. This environment was called Interactive C, and it made the contest much more accessible to students. The combination of a small, readily available board with an easy-to-use programming interface has made the 6.270 Controller Board, Rev. 2.21 C, one of the most popular controllers in use worldwide by hobbyists and educators.

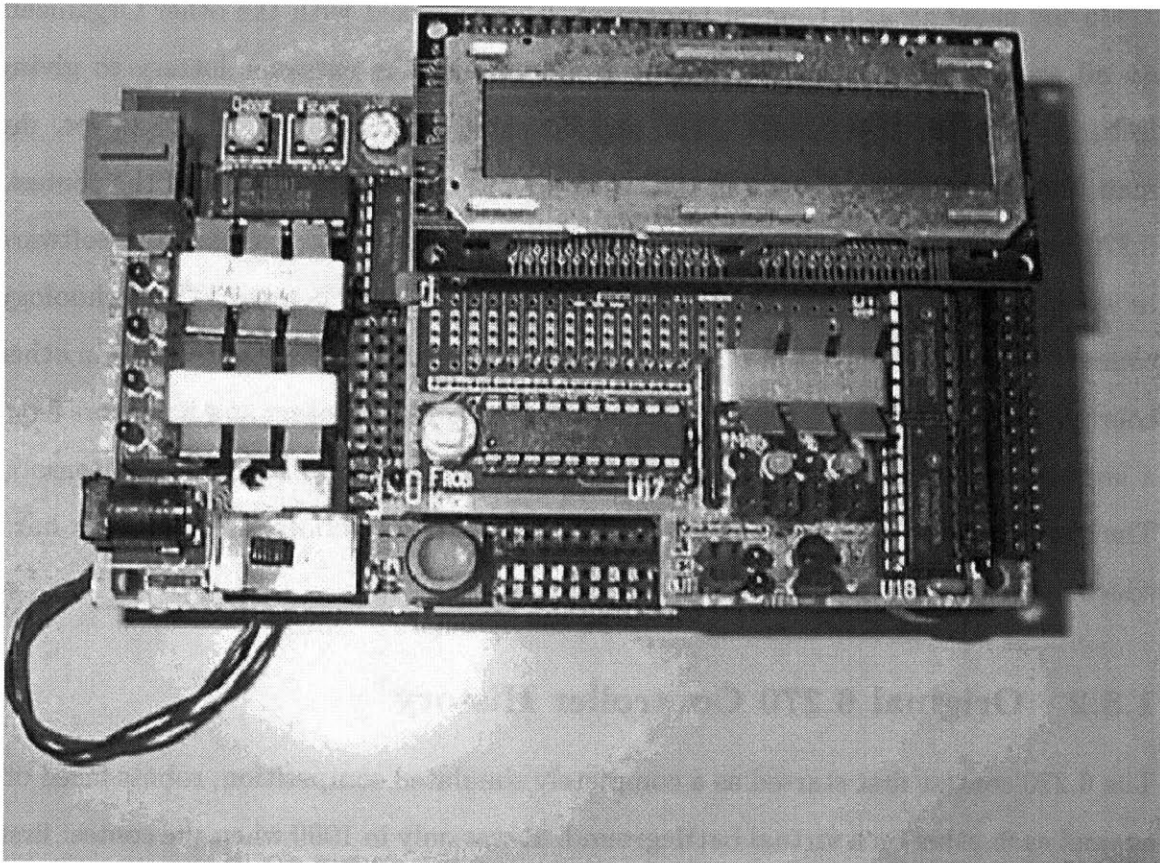


Figure 1-2: The Old 6.270 Controller

In 1997, an attempt was made by the 6.270 organizer staff to redesign the controller and take advantage of technological advances made since 1990. This effort produced a good list of areas for improvement and some early concept designs, but time constraints on staff prevented them from completing the design. In the Spring

of 1998, I decided to make a new 6.270 controller as a testbed for the ideas contained in this thesis. This decision gave me a concrete target for my ideas and a way to try out the BotKit concept in a real 6.270 contest. However, the amount of effort required to engineer the hardware design and work on the BotKit framework seemed prohibitive.

1.3.3 Compaq CRL Involvement

At this point, I presented the thesis proposal was taken to the Compaq Cambridge Research Laboratory (CRL), since they were my 6A sponsors. It turned out that CRL was interested in creating a small processor board for use in mobile computing research. After some discussion, CRL decided to help sponsor 6.270 by designing and fabricating controller boards for the 1999 competition. As a result, BotKit merged with the CRL Personal Server³ project.

1.3.4 Development Software

6.270's sponsorship by Compaq CRL took care of the hardware design work. However, the development software still needed some consideration. One of the reasons for the great success of 6.270 has been its interactive programming environment. This environment, called Interactive-C, makes development of code for robots accessible to beginners and makes development easier for advanced users. Many of the lessons learned from Interactive-C can be applied to embedded system software development in general, and any replacement environment for the contest must provide some of the benefits as well.

Interactive-C: Introduction

Fred Martin, one of the original organizers of the contest, spent some time analyzing the elements that make 6.270 special in his PhD. dissertation and found that a number of factors have contributed to its success. However, he noted that one of the major

³<http://www.crl.research.digital.com/projects/personalserver>

factors that drives the entire contest is the lack of time[17]. Teams have only three weeks to build and program a robot from scratch, and that introduces tremendous pressure on the part of software developers to deliver working systems quickly.

Using traditional development tools, the compile-link-debug cycle takes far too long in the 6.270 environment because the time spent downloading new code images to the controller board significantly slows down the process of refinement. Additionally, having to compile and link code just to run a few tests discourages experimentation and tinkering. The IC environment was created to address these issues by running a C interpreter on the target platform that integrated an interactive shell. Users could then type in C code over the serial line and watch it execute immediately to get feedback.

Advantages

This interactivity allows contestants to try out new algorithms and techniques without the overhead of writing a test program just for that purpose. Nothing beats being able to simply call a function with some test arguments by typing in the appropriate C code and pressing “enter”. For the majority of users, this is the single greatest strength of the Interactive-C environment, for it allows them to rapidly try out different parameters and test cases without resorting to writing test stubs just to exercise those functions. This shortens the development cycle considerably. The screendump in Figure 1-3 shows a session with IC.

Disadvantages

Unfortunately, one common complaint with IC is that it implements only a subset of the C programming language. In its current incarnation, programmers cannot use many C constructs that they take for granted like `switch/case` statements, `typedefs`, or function prototypes. This is not really a fault against the concept of Interactive-C itself, but against its status as a cult language. Development lags behind that of bigger vendors because it does not have the scale of support that a widespread development environment has. If there were a bigger user base for Interactive-C, this

```
C> printf("Hello World!");
Downloaded 9 bytes (addresses C200-C208)
Hello World!
Returned <void>
C> arrprint_2d(a);
Downloaded 9 bytes (addresses C200-C208)
1 10
5 50
-5 -50
2 20
10 100
Returned <void>
C> arrsort_2d(a);
Downloaded 9 bytes (addresses C200-C208)
Returned <void>
C> arrprint_2d(a);
Downloaded 9 bytes (addresses C200-C208)
-5 -50
1 10
2 20
5 50
10 100
Returned <void>
```

Figure 1-3: An example Interactive-C session

concern could be alleviated.

Also, the implementation of Interactive-C is not very portable. The interpreter is almost completely implemented in assembly language for the 68HC11 microcontroller family. This makes moving the environment to new hardware somewhat difficult unless the boards are also based on Motorola processors. With the new hardware coming online for the contest, this was viewed as the worst disadvantage of Interactive-C.

1.3.5 Final Software Specifications

With this analysis of Interactive-C in mind, a set of requirements was set out for the next generation development environment. Even though these requirements are specific to the MIT 6.270 contest, they can be extrapolated or modified slightly to encompass general embedded system requirements as well.

User Skill Level

The students taking 6.270 are generally MIT undergraduates in their freshman or sophomore year. Juniors and seniors do take the course, but fewer do because of the general increased academic workload by that time. As a result, 6.270 has traditionally drawn its entrants from underclassmen. Given the short time frame of the contest, one cannot expect the students to learn a completely new programming language that has no analogue outside of the contest. Interactive-C succeeded because it was at least somewhat like C. The replacement environment must continue this trend. In fact, it must be even easier than Interactive C, because compared to ten years ago when the contest first started a higher percentage of 6.270's entrants are not computer science majors .

Also, the environment should support both usual procedural programming and whatever new methods are introduced. Despite the advantages of state machine networks, one cannot ignore the fact that most users still feel more comfortable with procedural approaches. The goal of the environment is to make it possible to produce good code no matter what paradigms are used, not force users to use a specific one.

Platform Independence

MIT 6.270 robots are programmed through a serial connection to a host PC that runs custom terminal emulation software. This terminal software has been ported to a number of platforms that contestants use, ranging from HP-UX workstations to PC laptops running Linux or Windows 95. There is no easy way to standardize the platform all contestants work from since there are insufficient resources to

give all contestants workstations of one type, so platform independence for any new development system is a requirement.

1.4 Robot Controller Hardware/OS Requirements

The 6.270 organizers also decided that a replacement controller should be designed for the contest. The base level of functionality was defined to be at least that of the previous MIT 6.270 Controller Rev. 2.21. On top of that, it would have to fulfill requirements set by CRL so it could be used to perform mobile computing research. After some discussion, the key parts of the controller were as follows:

- Adequate processing power for advanced research applications. This included needs for networking, speech/video processing, and the like.
- Commodity software support. By this, we did not necessarily mean a commercial product like Windows CE, but we were trying to avoid having to write our own operating system specifically for this controller. This contrasts with the Interactive-C environment, which is narrowly targeted towards the Motorola 68HC11 family of processors.
- Expandability through industry-standard interfaces and connectors. The base controller board was meant to be just a testbed, so expandability was very important. We also intended to use standard 3rd-party expansion modules, so we stayed away from custom interfaces.

All in all, Personal Server (the name of the controller) was meant to be a simple, configurable computing platform.

1.4.1 Hardware

The core of the Robot Controller, the new 6.270 controller, is the Personal Server (see Figure 1-4) computing platform. The Personal Server's specifications are as follows:

- Intel StrongARM⁴ SA110 CPU @ 200 MHz
- 16 megabytes SDRAM
- 4 megabytes Flash Memory
- PCI bridge
- PCI to CardBus bridge and 2 PCMCIA sockets
- PCI to Universal Serial Bus (USB) controller and 2 USB connectors

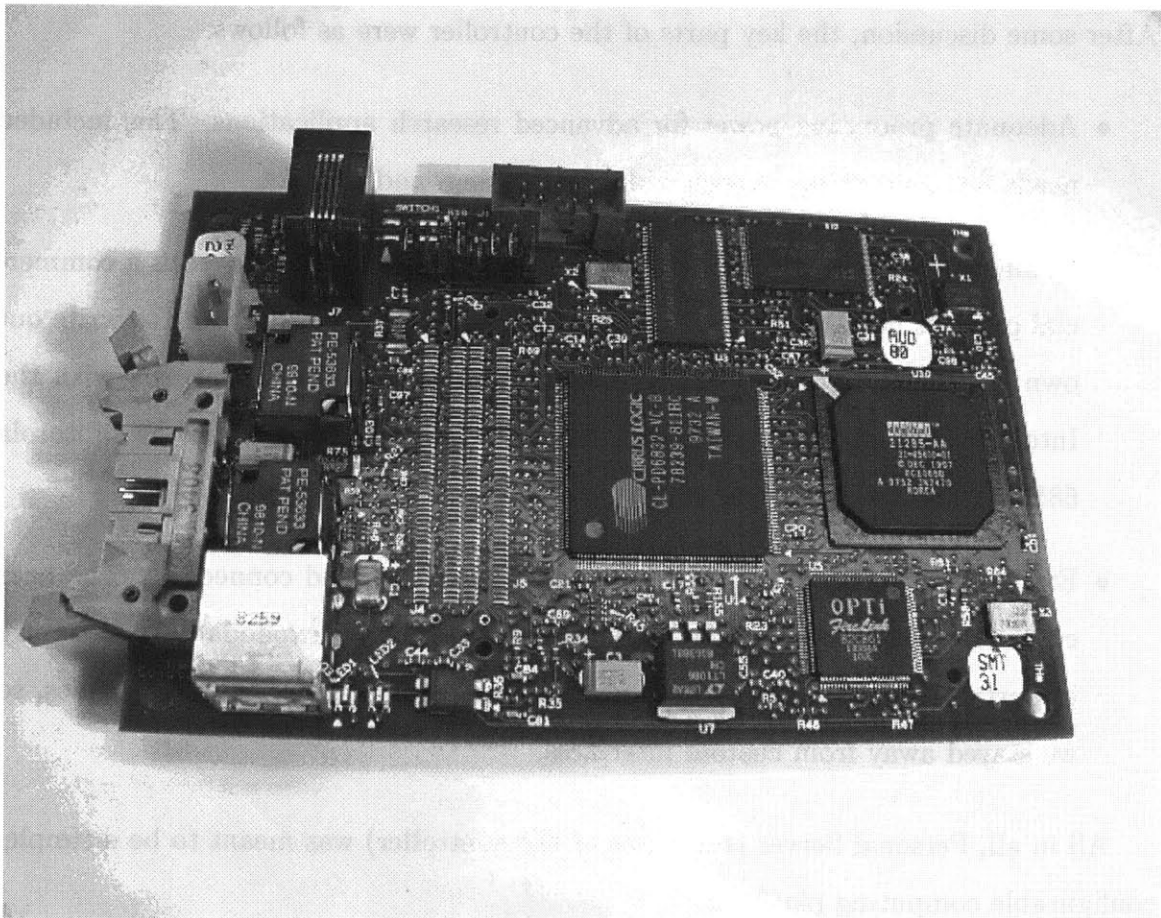


Figure 1-4: The Personal Server motherboard (3.6 inches by 5.4 inches)

For the purposes of the 6.270 contest, a daughtercard was designed to attach to Skiff as a USB peripheral and provide the I/O ports necessary for interfacing to

⁴StrongARM is a registered trademark of Intel Corporation.

motors, sensors, etc. This robot daughtercard (RoboDC, Figure 1-5) has the following on it:

- AnchorChips EZUSB 8051+USB core, 8 KB onchip RAM
- 10 H-Bridge outputs for driving motors (20 half-bridges)
- 32 analog inputs read through an analog-digital converter, 24 of which are accessible to the outside
- LCD connector and small LCD display

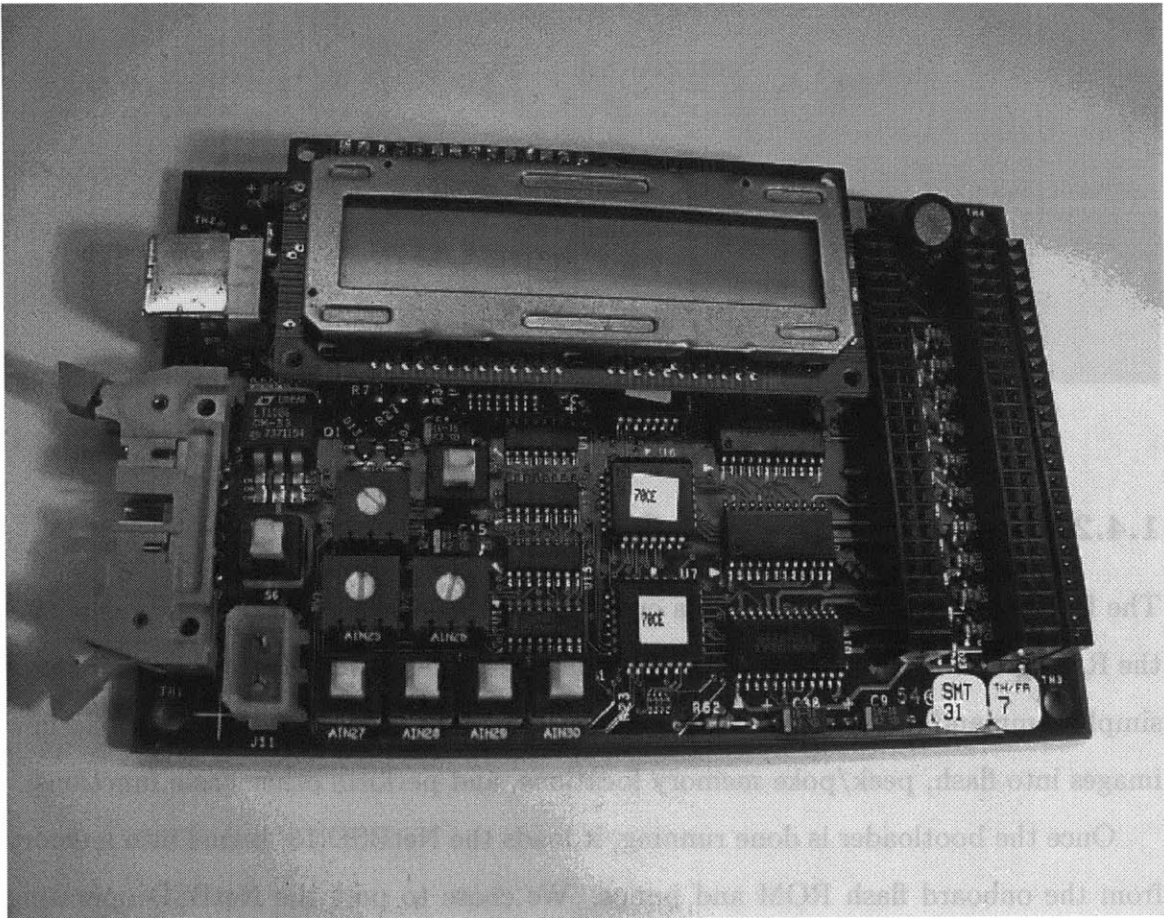


Figure 1-5: The RoboDC daughtercard

Together, the Personal Server and RoboDC are collectively called the RobotController (Figure 1-6). They both operate off of a 6 volt battery that powers both motors and the digital electronics.

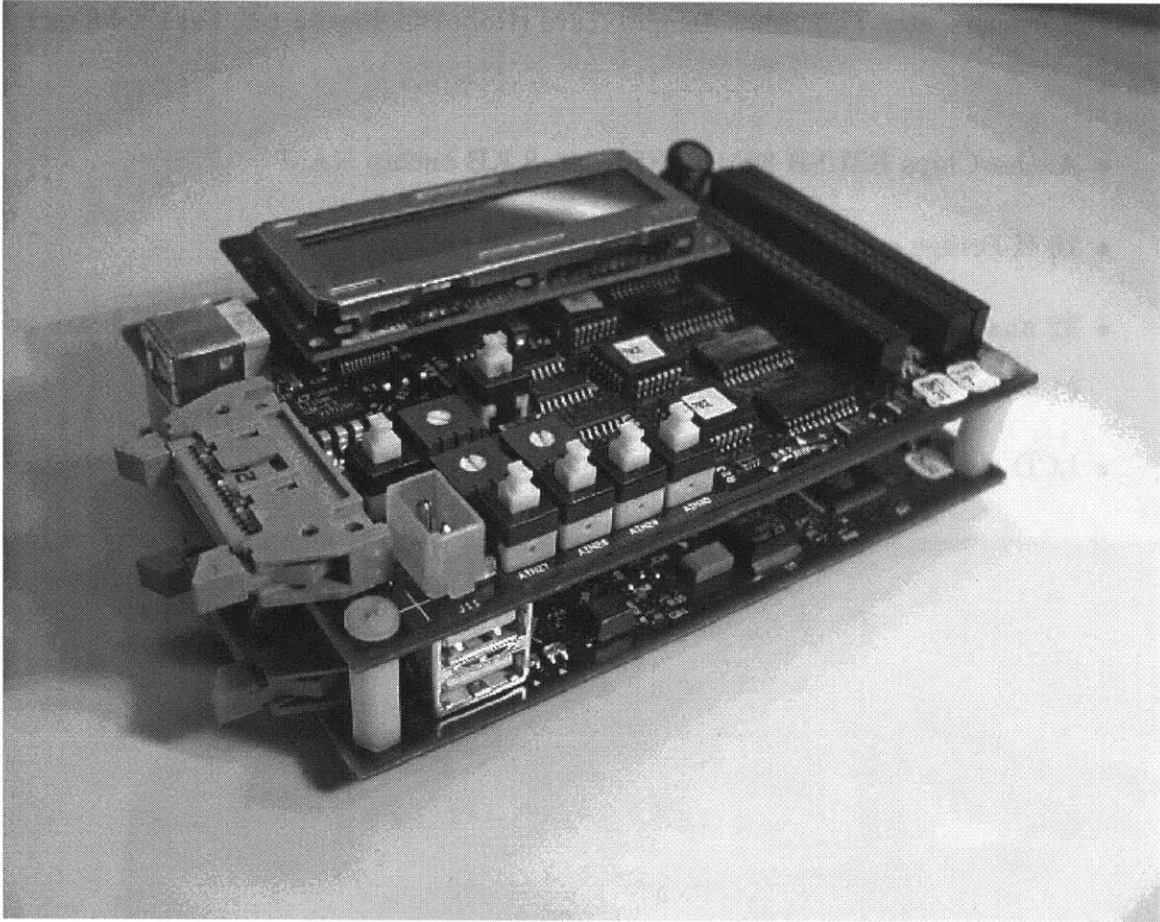


Figure 1-6: The assembled RobotController

1.4.2 OS and Support Software

The Robot Controller boots using a custom bootloader that takes care of initializing the RAM, PCI bus, and other miscellaneous tasks. The bootloader also implements a simple command line interface over the serial port that allows users to download new images into flash, peek/poke memory locations, and perform other basic functions.

Once the bootloader is done running, it loads the NetBSD[18] kernel into memory from the onboard flash ROM and jumps. We chose to port the NetBSD operating system to the Personal Server because of its open source code, portable codebase, and working USB support. We may consider other operating systems in the future, but for now NetBSD satisfies our needs very well.

The Kaffe Java Virtual Machine⁵ (JVM) runs mostly unmodified on top of NetBSD. This JVM has native methods with it to access the RoboDC daughtercard over the USB connection. The native methods appear to users as part of a class library that abstracts the actual hardware to users and presents a uniform interface.

The RoboDC daughtercard itself has an Intel 8051 CPU core on it, and we make extensive use of it to offload time-intensive tasks from the main processor. In particular, sensor polling, motor/servo pulse-width-modulation, and LCD control are all handled by the coprocessor. The firmware running on the daughtercard accepts commands over the USB bus and configures the hardware accordingly. Again, this process is transparent to the user; all the developer knows is how to call various methods in the RobotController API.

⁵Information on Kaffe can be found at <http://www.transvirtual.com>.

Chapter 2

Models of Computation

In this chapter we present the common problems and difficulties faced in writing robust code for embedded system and robots in particular. Then we compare the traditional programming models used for such applications, and then introduce the use of finite state machines. The advantages and disadvantages of each approach are evaluated so as to understand the design choices made in the final solution proposed in the following chapters.

Chapter 1 went over the basic challenges involved in designing and implementing an embedded system. However, the topic of software design has been left to this chapter because of its complexity. As previously mentioned, embedded system software has to be robust and interactive; the systems interact with the outside world at all times and must always respond. This stands in contrast to batch software which has all parameters specified before runtime and runs until completion with the assumption that parameters will not change until the run is complete. In an interactive system, events may occur at any time to interrupt, change, or even terminate the currently running code for various reasons, and the system must be capable of gracefully reacting to these arbitrary changes in inputs and make deterministic decisions about what to do next.

It is difficult to structure code in such a way that easing the enumeration of all the possible paths of execution easy to implement and handle is a difficult task. There are several different ways to model this flow of control, each with its own advantages

and disadvantages. In the 6.270 contest, many of these approaches can be seen being implemented by various teams with different levels of expertise, so this chapter will use 6.270 robot code as an example case.

2.1 The Line Follower

As an example, let us consider a robot that is driving along the playing field and is trying to follow a black line painted onto the table. This line is for the most part straight, but may have curves or even 90-degree turns in it. For the sake of realism, let us also say that the purpose of following this line is to collect balls on the line's path and score points.

The robot has two drive motors, one for each side (left and right). In this way, it drives much like a tank – to turn left, just run the left motor at a slower speed than the right motor, and vice versa for turning right. For sensors, the robot starts with three light-sensitive detectors that can “see” a black or white surface. We can assume for simplicity's sake that some preprocessing code takes the analog sensor reading and turns it into a digital “white” or “black” reading with a fair degree of accuracy. These sensors are placed in a row such that if the robot is centered over the line, the sensors completely span the line as shown in the diagram (see Figure 2-1).

Let us assume that the robot starts centered on the line, *i.e.*, the two outside sensors report that they are on the white table, but the middle sensor is on the black line. Now, the task is to follow the line for as long as possible and scoop up balls with the aid of a vacuum-cleaner-like attachment to its front.

There are many possible ways to solve this problem, but the most common algorithms are essentially feedback loops that correct the speeds of the motors on each side of the robot to keep positioning the sensors on top of the line at all times. If the left sensor falls off the line, a slight adjustment to the right is made, and so on. Of course, in practice this problem is much more complex than it seems. In the following sections, we will attempt to implement this algorithm using different methodologies and expose the most common pitfalls for each.

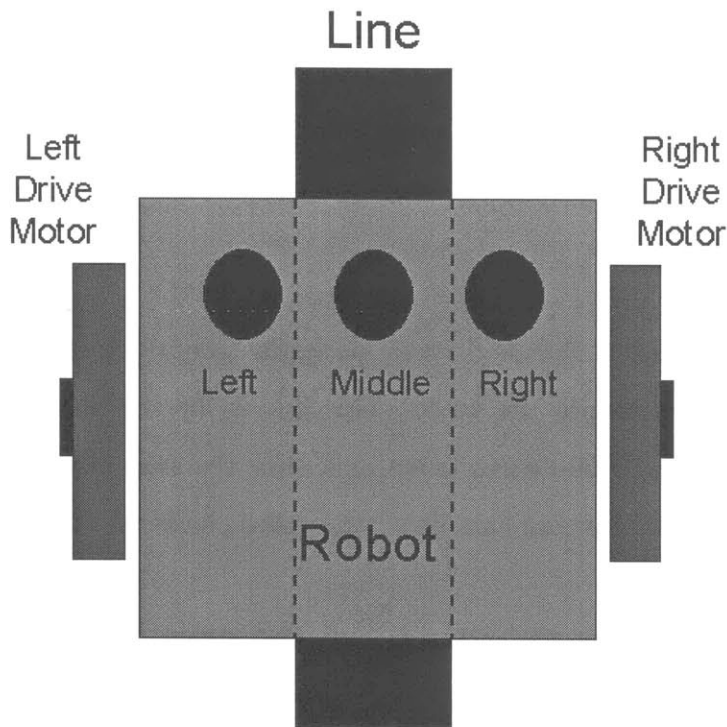


Figure 2-1: Note the position of the sensors such that the outside sensors are on the white surface while the middle sensor is on the line.

2.2 Procedural Models

Most introductory programming classes start students off with basic concepts of program flow. They show how to break complex tasks into procedures, and also introduce concepts like loops, conditional statements, *etc.* Most software engineering classes tend to teach students how to write batch-mode programs. In other words, the programs normally assigned for homework are structured such that they read in some input, operate on it, output the result, and exit. Dividing such a program into procedures helps to reduce the apparent complexity of the code and increases reusability.

This approach is easy to learn and works well for the majority of projects. Splitting the pieces of a program into procedures and establishing a clear flow of data through the program from procedure to procedure definitely makes for a legible diagram. However, if the number of distinct code paths is high, the system can get confusing. This increases the complexity of the implementation and the possibility for mistakes

and necessitate more time spent in debugging the code.

2.2.1 Implementing the Line Follower

Using conventional methods, one might implement the line follower using the following algorithm (Figure 2-2). A variant of this code snippet is introduced in the first lecture of 6.270 every year as an introduction to feedback loops. It has been our experience that most students can understand a piece of code like this within a fairly short span of time, but more complex examples tend to make people miss the overall idea. Hence, we start with this implementation.

2.2.2 Analysis

This algorithm revolves around a fixed feedback loop in the center that continually checks the state of the sensors and makes adjustments to the drive motors based on sensor results. Even in this naive form, the algorithm actually does work, and a robot programmed in this manner is capable of following a reasonably straight line down a table. The code is still fairly easy to understand as well.

However, throw in a more complex situation and the problem space exponentially increases. For example, the above example has only one degree of correction – no matter how far the robot is off the line, the code will only rotate the robot at a certain rate. If the distance being traveled is large, that is acceptable, but for a short trip this feedback parameter does not work. The robot will run out of room before it gets back on the line. However, one cannot simply turn the feedback rate up, because then the robot will likely over-correct for minor course deviations and proceed to oscillate back and forth across the line.

The solution to this demands some sort of proportional feedback mechanism that varies the amount of correction based on how far the robot is off course, and how quickly it is traveling. The robot's speed is important because the rate of correction is also heavily dependent on movement speed – steering input is amplified much more at high speed.

```

int line_follow(int timeout) {
    int i;
    int rsens, csens, lsens;
    for(i=0;i<timeout;i++) {
        /* Assumes sensor threshold is compatible
           with a digital sensor's. */
        rsens = digital(RSENS);
        csens = digital(CSENS);
        lsens = digital(LSENS);

        if(csens && !rsens && !lsens) {
            /* On Line */
            drive(100, 100);
        } else if(lsens) {
            /* Turned to the right. Turn back left to correct */
            drive(20, 80);
        } else if(rsens) {
            /* Turned to the left. Turn back right to correct */
            drive(80, 20);
        } else {
            /* None of the sensors read a line. Return a 'lost' signal */
            return 1;
        }
    }
    /* End for(i=0;...) */
    return 0;
} /* End line_follow */

```

Figure 2-2: This fragment is presented to 6.270 contestants on the second day of class every year. Thanks go to Robert Blau (robblau@mit.edu) for contributing this version.

Implementing this improved feedback mechanism within the existing framework is possible; the result is significantly more complex and harder to read, but the basic flow of control is the same. However, this code still assumes that the robot is not doing anything else while it is moving down the table, and furthermore, that following the line is a goal to be pursued without caring about other external events that might occur before the line ends.

In real life, that sort of idealized situation rarely happens. It could be that the line is just a guide, and the robot is supposed to stop after a certain distance and

turn off the line or do something else. The procedure could be modified to check the distance traveled each time through the loop and return once that distance has been traveled. But what if the robot never makes it there? It could get stuck, or the wheels might spin on a slick surface, or any number of things. Hence, a timeout mechanism is needed. Additionally, a 6.270 contest table is rarely that simple to navigate. Paths on the table are often obstructed by blocks, lines are rarely perfectly straight, and the sensors themselves are usually not optimal.¹ More code is added to the main loop to check for a time-elapsed function, and the list goes on...

The end result of all this is that the original line follower procedure has gained so much complexity and extra functionality that it has become hard to follow. This makes it harder for a developer to make additional modifications or debug the code, and greatly increases the likelihood of unforeseen test cases.

Most 6.270 contest robots have been coded this way since the contest was started. In our experience, this methodology is easy to learn at first and works well when contestants are preoccupied with learning basic parts of robot control, but when integration into a contest-ready piece of software occurs everything tends to break all at once. It seems that the main problem here is the difficulty involved with controlling multiple inputs and outputs on a robot with only a single flow of execution. Single-threaded systems must by necessity visit many different sections of code one at a time and take care of business before going on to the next procedure. However, this makes the systems difficult to code and debug.

2.3 Multithreaded Procedural Code

One way to reduce the complexity of a big monolithic control loop is to split the relevant sections into their own loops and run them simultaneously. This requires the code to become multithreaded either preemptively or cooperatively. However, the overall complexity of the system increases and requires higher effort to implement

¹It probably would help if we did not have to buy them at the surplus store. If anyone reading this thesis would like to help the contest, please send money so we can get better stuff.

and debug, as is shown in following paragraphs.

Implementing this scheme helps by allowing multiple threads of execution to run at once. There is no large “main loop” through which all execution flow has to pass, and the individual loops that run by themselves are easier to code and understand. In the case of the line follower example, the code controlling motor movement can now run independently of the code that checks for collisions or monitors timeout conditions.

This solution seems ideal at first, but in real life it has proven to be very problematic. Multithreaded programming is a very difficult task. This barrier has kept many developers from taking advantage of this paradigm. Having to keep track of what different threads are doing at the same time and coordinate interactions between threads using primitives like message queues and spinlocks requires an extremely high learning curve for most developers. This barrier to entry makes it impossible for most 6.270 contestants to use multithreaded code properly, especially since more and more contestants are entering with no previous programming experience. Mistakes are common even for experienced developers, and bugs are sometimes almost impossible to find because of the many intricate cases that may occur at any point during runtime to crash a multithreaded program.

Anecdotal evidence of the problem with multithreaded code can be found in 6.270’s history. 6.270’s Interactive C programming environment has had support for multithreaded functions since its debut. The default code for robots that the organizers supply to contestants includes one thread routine to automatically shut the robot down after 60 seconds. Many teams each year try to take advantage of even more threads, but most of them revert back to single-threaded algorithms by the time they submit their robot for final competition. A rough estimate gathered from the past three years of contest code shows that of 150 teams, only about 40 ended up entering multithreaded final contest code (excluding the built-in start/stop thread provided by organizer staff).

Viewing a complex interactive system in terms of separate procedures generally causes problems regardless of a single or multi-threaded implementation. The diffi-

culty inherent with expressing embedded system behavior stems from the multiple paths control may take through the same code. Accounting for all these paths and expressing the flow of control in what is essentially still a single-path paradigm is difficult because there is no clearly defined begin and end state in these systems.

2.4 Finite State Machines

One way to reduce the confusion is to outline the different code paths before implementing them in code. A state diagram is one such method used to model a system and compress multiple combinations of inputs and outputs into a succinct representation. By presenting the different positions of the system in various combinations and enumerating how the combinations can lead to each other via different types/numbers of inputs, a table emerges that effectively captures all the possible states of the system.

A formalized visualization of this concept leads to finite state machines or FSMs. A FSM is most simply described as a set of input events, a set of output events, a set of states, and a set of transitions. Consider the example shown in the following Figure 2-3. This diagram shows an FSM with input events [a,b] and outputs [u,v]. Each node represents a state, and the connections (arcs) between states represent transitions[10]. State machines have to start in some initial state, so the arc without a source state pointing to the initial state does not actually represent a transition, but is more of a pointer.

There must be a way to determine whether to take a transition or not. This takes the form of a boolean expression over the input events that is attached to each transition. The expression is called a guard. When the expression evaluates to be true, that particular transition is taken, the current state shifts to the destination of that arc. For some transitions, there is an action/output when they are taken. These actions are attached to the guards.

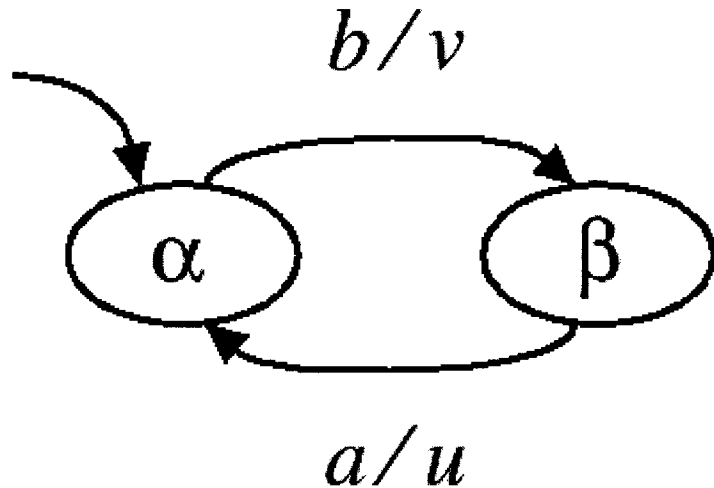


Figure 2-3: A simple finite state machine. This diagram comes from Girault, Lee, and Lee's paper on hierarchical FSMs[13]

2.4.1 Advantages

The main advantage of a finite state machine is its ability to represent many possible execution paths in a compact manner. Each state may have its own table of transitions, with their own trigger guards and actions, and this helps to encapsulate all possible behavior in a specific realm of the problem space. By isolating the system into a network of states, one can focus on a particular state and its relationship to other states without having to worry about the others. In practice, this makes implementation much easier because it segments the system by states, and states can be added or deleted from the system piecemeal without too much effort. All one must do is follow all transitions coming into and leaving that state and modify the tables in the connected states accordingly.

Most finite state machines are implemented by means of a core “event loop” that continually reads the inputs and updates the current state accordingly. This core loop is easy to understand, and more importantly, does not need to be modified a new state is added to the system. All of the complexity in a finite state machine is hidden in the initial setup, so understanding the flow of execution is easy once the state diagram is drawn. This gives FSMs an advantage in development over the mess of nested conditional statements that characterize procedural code.

Going back to the line follower example, Figure 2-4 exemplifies how one might convert the system into a state diagram. The inputs to this state machine are the sensors, and the outputs are the commands sent to the drive motors. One can enumerate the possible states of this system through noting that staying on course means trying to remain in the state where the sensors straddle the line. Starting from there, the transitions which can occur (veering left or veering right) are easy to come up with. For most people this diagram is much easier to understand than the raw code or even flowcharts that the procedural methods produce.

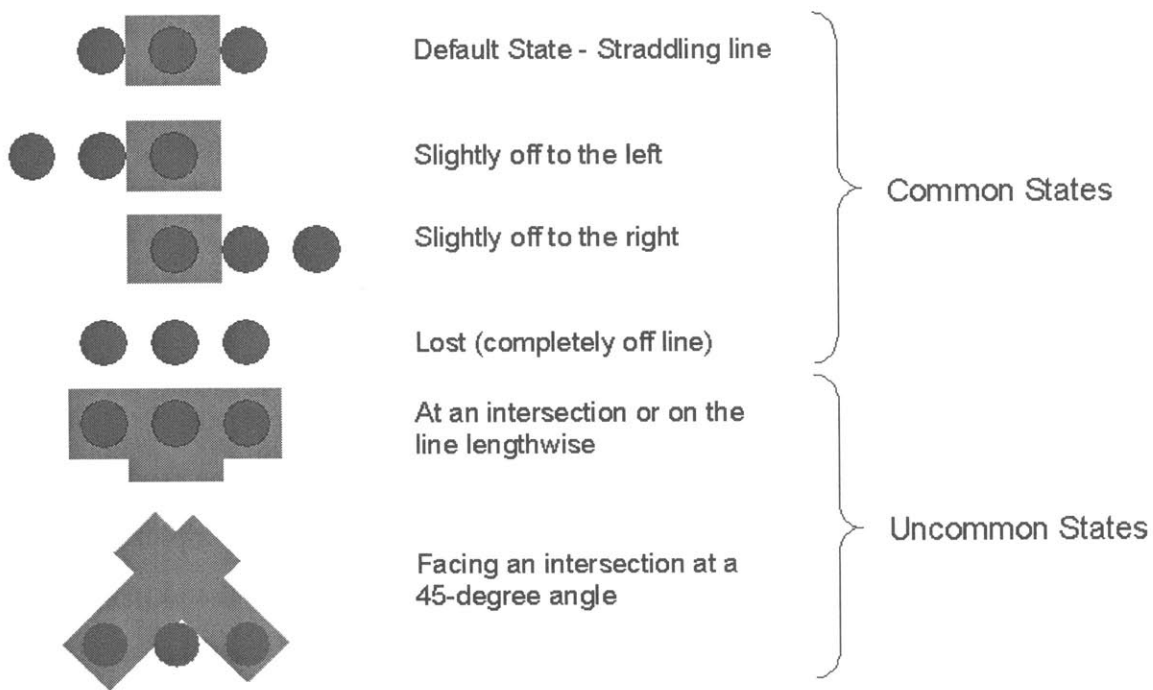


Figure 2-4: Some of the possible states the sensors can be in for the line follower. Note that two of the states in the figure correspond to odd situations that would be hard to catch otherwise if one were to just start writing the code without thinking the problem through first.

2.4.2 Disadvantages

However, FSMs are not without their drawbacks. FSMs are easy to implement, but it is not quite so easy to come up with the state diagrams in the first place. Reducing a system down to a collection of states requires some thought and is somewhat difficult to do for an inexperienced developer. Most computer science students are taught to

break problems down into smaller pieces and solve them using procedural methods; they have less experience breaking problems into state instead.

Experience with the MIT 6.270 competition has shown that finite state machines work well in the classroom with pre-meditated examples and clearly articulated systems, but when contestants attempted to use the same principles in real robots the state diagrams got extremely large and unwieldy. The problem is that for a single task, state machines model the system extremely well. However, there can only be one active state in a state machine at any given time. This makes modeling of multiple concurrent tasks difficult; the FSM must rapidly transition between states relating to the different tasks to keep them all moving at the same time. As a result, the FSM methodology exhibits the same problem that single-threaded procedural code does, even though the initial modeling step is easier.

2.5 Multiple Concurrent FSMs

The natural extension is to run multiple FSMs in parallel using separate threads of execution. Running multiple FSMs allows the individual FSMs to be fairly simple; yet, having the FSMs work in concert allows the system to express complex behavior as if it were run using a single very big FSM. Keeping the pieces small and in separate FSMs reduces the effort necessary for users to integrate various portions of their code together and helps solve the concurrency issue raised by modeling multiple tasks.

Going back to the line follower example, let us now try adding more complex behavior to the system. If we also want to track how far the robot has traveled and take action based on that, more guards and actions could be added between the existing states. However, this makes the input and output vectors get larger and more complicated. Instead, a separate state machine that has only distance measurements as inputs makes the picture much simpler (see the Figure 2-5). The output of this second state machine then becomes an input to the first, linking the two FSMs together. A network of FSMs now exists that exhibits more complex behavior than the original, but the original FSM has not changed or gained complexity.

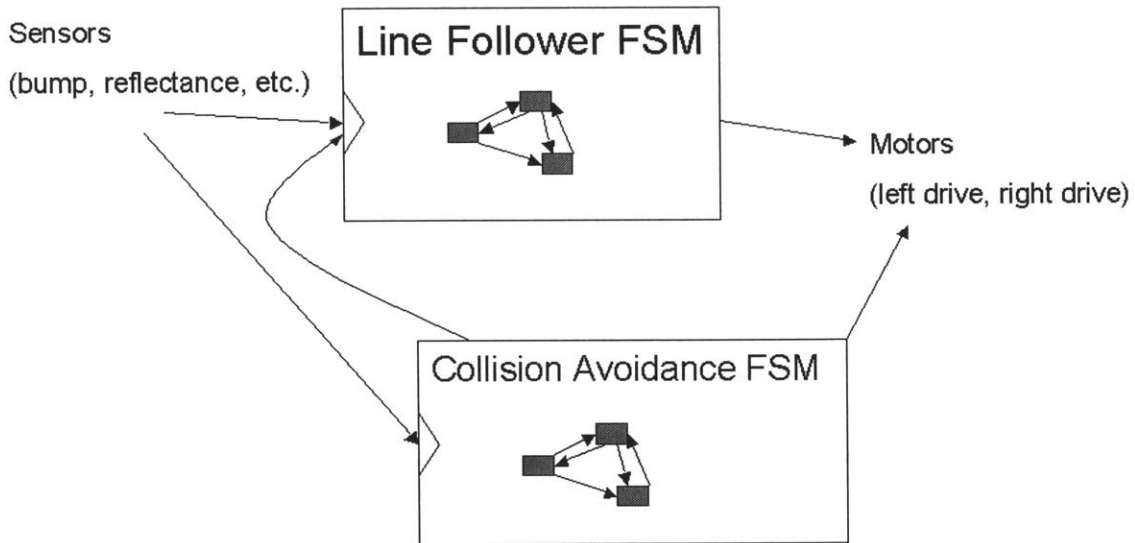


Figure 2-5: An example FSM network

Networks of FSMs have recently come to the fore as a promising new way to model reactive systems[12]. A lot of recent work in this field has come out of the Ptolemy Project[8] at the University of California, Berkeley. In particular, an in-depth look at the combination of finite state machines with multithreaded environments produced a working system that is capable of modeling complex real-time systems, particularly in the field of digital signal processing[1]. Prior attempts to take advantages of FSM networks are documented in papers on robotics, real-time operating systems, and other applications.

However, running multiple FSMs concurrently raises the issue of complexity from a different perspective. If all the different state machines operate independently of each other and have no need to communicate, the implementation is simple - just set up the different threads and let them run. But if the FSMs depend on each other, some sort of inter-FSM communication mechanism is required. I attempt to address this issue in Chapter 4.

Chapter 3

The Subsumption Model

In the previous chapter, we went over various ways of breaking embedded systems into manageable pieces and implementing robust control loops for them. In this chapter we present the solution arrived at for use in BotKit, which makes use of the subsumption architecture[7].

3.1 Introduction

Finite state machines ease the task of expressing multiple execution paths, but developers still face the task of deciding how to model a system. Real-world systems often do not easily break down into networks of finite state machines because the states are heavily obscured by minute details of interaction. A way of viewing a system that encourages decomposition into smaller systems was needed so it could be applied to the robot contest.

Rodney Brooks from the MIT Artificial Intelligence Laboratory ran into much the same problem in the course of his own research in robotics during the early 1980s. Brooks chose to decompose systems based on observable external behaviors (see Figure 3-1)[2]., Brooks chose to instead decompose the system based on its observable external behaviors[2]. This results in a horizontal breakup of the system into layers that completely span from raw sensor inputs to observable outputs, which stands in contrast to traditional models that split the task vertically and insulate layers from

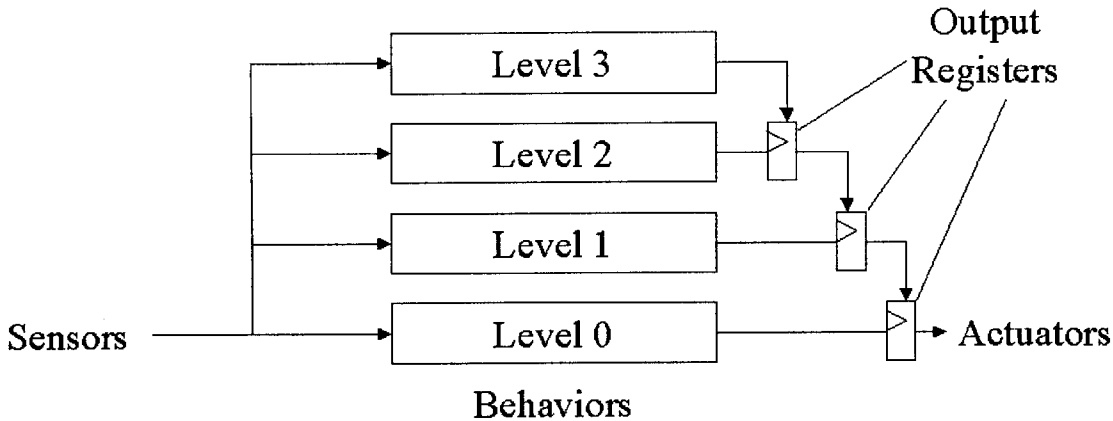


Figure 3-1: Layers of control are stacked horizontally so that lower-level layers by default have control unless higher-level layers take over. Output registers combine inputs from the multiple layers and either sum or choose inputs to present what goes to the outside actuators and motors. Note that one can take off the higher layers and still have an operational system. This figure is taken from Brook’s paper “A Robust Layered Control System for a Mobile Robot” [2]

each other through abstraction barriers.

Brooks’ contention was that breaking the systems up horizontally allows one to concentrate on making a robot perform a specific behavior and do it well, then incrementally add functionality in the form of more specialized behaviors. Additionally, each higher level of behavior includes as a subset some or all of the lower levels. Since higher levels include lower levels, the higher level behaviors can be seen as a top-down enforcement of design constraints on the lower levels, providing an overall structure to the program[15].

3.2 Advantages

This architecture emphasizes the independence of each behavior – developers create each behavior and complete it before moving on to the next. By debugging lower-level behaviors completely and then leaving them alone, higher-level behaviors are able to assume some base level of functionality before modifying the observable outputs to create more complex actions. In the event that higher-level behaviors are incorrect or produce output at the wrong times, the higher-priority lower level behaviors will

still produce mostly sensible results[4]. Essentially, the low-level behaviors step in for the system states that are undefined or poorly handled by high-level constructs.

Returning to the line follower example, let us attempt to break this problem down using behaviors (see Figure 3-2). The most obvious behavior in the system is the robot's attempt to stay on the line. The inputs to this behavior are the light sensors under the robot, and the outputs are the motors. This behavior is the default one given normal operating conditions. But as previously noted, this system must be capable of handling adverse conditions enroute to accomplishing the primary goal of following the line.

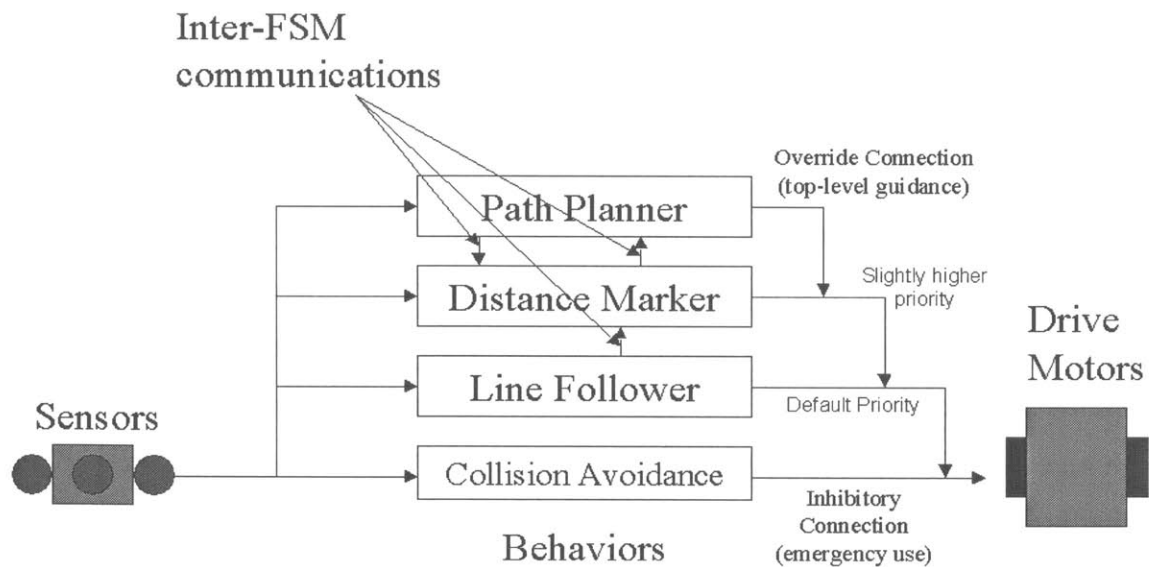


Figure 3-2: Note that the base behavior just has the robot avoid hitting walls, and at the top level a path planner picks which lines to follow. Behaviors also communicate directly with each other to exchange state and data.

The adverse cases can be controlled by other lower-level and high-level behaviors. For example, a base behavior may want the robot to avoid getting hung up on walls, so its response to bump sensors is to back away. Since this behavior is important for robot safety, one may set the priority of this behavior somewhat higher than the others. By having this behavior run concurrently with the line follower behavior, a combination of the two behaviors is achieved with minimal effort to actually link the two. Note that each of the two behaviors operates independently; the only link

between them is the ability of one to override, or inhibit, the actions of the other when push comes to shove. This sort of behavior layering leads to a more robust system because developers can worry about high-level thinking at one level and safely assume that base cases are still covered by lower-level behaviors[6].

Another positive side effect of modeling systems in this way is that sensor inputs are exposed in their raw form to all levels of behaviors. This allows each behavior to treat sensor data in a way that is useful to it without having to worry about other behaviors' needs. This contrasts with the approach of encapsulating all sensor input into one procedure that has to provide relevant interpretations of data to all layers above it. Such an approach leads to an overly-general and non-optimal solution for all the layers on top because no one solution can be perfect for all.

Additionally, each behavior layer has optional direct access to the system inputs and outputs. For those outputs that need to be shared among different behaviors, output registers can combine outputs from multiple FSMs and decide how to present the results to the actuators. Furthermore, these registers need not be present at the very end of the loop; they can be interspersed between different layers and handle very specific situations. This solves the problem of implementing a generic signal "combiner" at the end of the decision cycle needed in other schemes. With a vertical separation of inputs/outputs away from other layers, the code needed to process inputs for one layer must be fitted to also meet the requirements of other modules, and this leads to overly general abstractions that are not useful. Using subsumption allows the input/output processing to be tailored to each layer without worrying about side effects on other layers. However, communication between layers is not hindered, because the behaviors can still influence each other by connecting outputs from some layers to inputs on other layers.

3.3 Implementation

3.3.1 AFSMs

The initial implementations of the subsumption architecture were performed at the MIT Artificial Intelligence Laboratory. In their mobile robot testbed they implemented the different behaviors as finite state machines running on multiple small processors connected through a network of wires. Inputs to the processors came either from connections to other processors or directly from sensors[3].

One notable trait of this setup is that no global shared memory is needed between nodes – whether the system was running as multiple threads on one processor or one thread per processor is a moot point. This greatly eases implementation because no means of explicit synchronization between nodes is assumed. Together, this collection of processors, wires, and inhibition/override mechanisms has been termed an Augmented Finite State Machine (AFSM)[6] by Brooks.

3.3.2 Alternatives to AFSMs

The subsumption architecture does not necessarily need to be implemented using AFSMs, however. All the known implementations of the architecture do make use of finite state machines and connections of some sort between them, but one can just as easily imagine substituting a normal threaded procedure in place of a FSM and having it interact with the rest of the behaviors in a subsumption-based system. After all, one of the chief goals of subsumption is to not make behaviors depend on each other or force them to know details of each other's implementation. Behaviors are black boxes, and therefore finite state machines may or may not be used to implement them. The BotKit framework takes advantage of this abstraction to allow procedural code to interoperate and mix freely with pure AFSM implementations; details of this are presented in Chapter 4.

3.4 AFSM Disadvantages

3.4.1 Learning Curves

The main issue with subsumption, AFSMs, and alternative embedded programming paradigms in general is inertia. Specifically, most programmers are already used to the procedural development styles and are reluctant to change. The learning curve associated with switching to finite state machines and subsumption is high for most people because they have not had to think that way before. As a result, many users find it difficult to get used to jump whole-heartedly into the FSM paradigms.

Part of this barrier cannot be helped; finite state machines emphasize a flow of program execution that runs counter to most programmers' idea of programming as a series of batch processes. Similar obstacles were faced with the introduction of event-driven systems in modern graphical user interfaces like the X-Window system, Microsoft Windows, and the Macintosh OS. One major goal of this thesis was to find a way to ease the transition from a procedural-based paradigm to a behavior/state-machine based one. The subsumption architecture is part of the answer.

3.4.2 Small Pieces

Lastly, for small systems, procedures are quick and easy to implement given the programming languages in common use today. C, Java, and other mainstream languages have all evolved features and constructs that make it easy to quickly write procedures. In contrast, small ideas that involve step-by-step sequences are generally cumbersome to implement using state machines. For rapid prototyping and small behaviors, a quick procedure to handle the small number of possible states is actually faster to implement.

Given the abilities of contestants in the 6.270 contest, this distinction is important. Most 6.270 robots are programmed incrementally – contestants learn to teach their robots simple tasks from day to day and combine them to create intelligent routines near the end. Each of these tasks by itself is simple, but combining them with a

procedural approach is difficult because of the multiple code paths as outlined in Chapter 2. Additionally, some systems have so many complex states that modeling all possible inputs/outputs causes an exponential increase in the number of transitions between states. Enumerating the table by hand becomes impractical, especially since many of the possible input combinations are variants on the same topic. It would be simpler to express these transitions in terms of short procedures that allow active processing of inputs to be performed before choosing a destination state.

3.5 AFSM Wrappers for Procedures

One possible solution is to allow AFSMs and small procedures to be intermixed in the same system. This is the premise behind BotKit – that a framework allowing procedural code to interact with FSMs running in their own threads allows both bottom-up incremental programming and top-down integrated design. This is done by wrapping procedural code in a layer that provides the same input/output paths as those used by AFSMs. For example, the special cases that make up the line follower can be implemented as separate procedures that run in their own threads. They effectively act as behaviors in their own right, and can manipulate, inhibit, and process inputs and outputs from the other FSMs in the system just like any other AFSM.

Now users have the ability to write short procedural code snippets to perform very simple, small tasks, and they can reuse, and more importantly, combine them later in networks to implement the subsumption architecture. High level behaviors that are more complex can be better stated by a finite state machine, but they are still capable of interacting with the smaller procedural pieces. This allows a working system to be brought online very quickly, but does not create the impression of a gross “hack”. The procedural components can be left there or replaced with finite state machines later on.

3.6 Forming BotKit

With these piece in place, a framework was designed to take advantage of the subsumption architecture and AFSMs, while retaining the ability to run standard procedural code inside. This framework ended up being called BotKit: The Robot Construction Kit largely for historical reasons. The original project, which was started three years ago to produce a replacement robot controller, had the goal of producing a “kit” of interchangeable hardware and software robot component for 6.270 contestants.

Chapter 4

Implementation

The BotKit framework consists of a Java class library. These classes help botKit users quickly construct AFSMs running in separate threads that communicate through registers and combine behaviors. This section describes the choice of Java as the implementation language and runs through example code to demonstrate how the subsumption architecture and AFSMs are combined with procedural code fragments to produce a robust embedded system. The deployment of the Robot Controller and portions of the BotKit framework in the 1999 6.270 contest is also described.

4.1 Why Java

4.1.1 Drawbacks of C

The C language has been used in 6.270 for many years. More existing work would be preserved if it were again used for the implementation of BotKit, but several factors prevented C from being chosen. First, C lacked sufficient support for encapsulation of the internals of FSMs. This lack of encapsulation failed to satisfy the ease of use requirement. The idiosyncrasies of the C programming language are also fairly difficult for beginners to master well in three weeks, and that was a heavy factor in the decision.

BotKit makes heavy use of multithreaded code to implement the FSM networks.

C has no language support for threads – it is all contained in libraries. Dealing with C’s add-on thread support threatened to delay the completion of the project, so a language with better thread support was needed.

Lastly, writing the BotKit framework in C would require users to have C compilers of their own on the host machines. Running a C interpreter on the target controller was an option, but that was problematic due to the complexity involved in a writing and maintaining a C interpreter for years to come. The problems inherent to Interactive-C (inability to provide all C features, subtle language differences) would also be present. In the absence of an interpreter option, cross-compilers would have to be ported to each supported architecture and maintained for users to develop with. The maintenance headache this would cause was deemed unsatisfactory.

4.1.2 Portability

The Java programming platform attempts to solve many of these problems by using a virtual machine to implement its services. First, the virtual machine allows the code to be taken from platform to platform without recompiling. This is not such an important advantage for an embedded system because these systems are generally not meant to run code from other sources often, but it does open up many possibilities in the realm of simulation. Running a virtual machine means that users can run an emulated version of the system on their desktop system and test the code before they bother downloading it to the target platform.

Additionally, Java implementations have been completed for most major platforms in existence, and therefore the compiler support is already built-in. Using Java saved much time in development of cross-platform tools and allowed the author to focus on the AFSM libraries instead.

4.1.3 Stronger Typing and Garbage Collection

From an academic standpoint, Java is also much cleaner than the C programming language. Java has stronger typing and therefore encourages developers to think

more carefully about their code. The lack of untyped pointers also automatically eliminates most common programming errors and lets users focus on the code at hand instead of memory corruption. Thanks to Java's built-in garbage collection, memory management is left to the environment, which greatly eases the task for beginners.

However, Java does still retain much of the look and feel of C, and that is a major advantage when teaching it to students who may have had experience with C but not Java. In the years to come, the makeup of students coming in with experience with either language will likely shift towards Java instead of C, so this factor will be less of an issue.

Object-Oriented Programming

The object-oriented nature of Java also offers opportunities for a clean BotKit implementation compared to C. The BotKit implementation uses object-oriented features to encapsulate the running of procedural code inside of finite state machines. Users of the BotKit library also gain some measure of code reusability if they choose to use the BotKit framework and extend its base classes. For example, a user may decide to create a new type of FSM that sets up commonly used states for a specific application. To do so, one only needs to inherit from the base FSM class and add required functionality, and then this new "pre-built" FSM is available to the entire system.

Using an object-oriented language also allows useful abstraction of the hardware interface layer away from users. This was used extensively in the the implementation of BotKit for the Robot Controller controller board to hide implementation-specific details from 6.270 contestants. In the future, hardware changes in the controller platform should do little to affect the code running on top of them so long as the wrapper classes are designed properly.

4.1.4 Threads

Java's built-in support for threads is a major design win for the BotKit implementation. The language directly supports thread synchronization and exclusion, which makes it much easier to implement support for BotKit's networks of FSMs. The ability to easily mark regions of code as exclusive also made implementation and debugging of the framework easier.

Users of the BotKit framework do not have to worry so much about multithreaded programming issues since all aspects of inter-behavior or inter-FSM communication are handled by the library. AFSMs operate independently of each other and do not block waiting on inputs since they are always in one state or another, so deadlock issues are for the most part avoided. Another type of deadlock can possibly occur where a FSM never transitions out of a certain state because there are no outgoing connections to other states, but that design error is easier to catch than a thread deadlock situation; a look at the state diagram will quickly show the lack of outbound transitions.

Additionally, communication between the various FSMs in the network is allowed only through unidirectional registers. These registers buffer data flow between FSMs and have logic of their own to handle multiple inputs either autonomously or with user-specified actions. At no time is there no output coming out of the register; the default inputs may be overridden or inhibited by higher-priority FSMs, but there is always some kind of output value exiting the register. It may be wrong from an end-user perspective, but it should not put the system into a wedged state. Finally, the registers are all clocked at the same time – the FSM network runs a continuous loop that updates all register contents independent of AFSM execution. Having registers update simultaneously gets rid of many potential timing issues that plague other completely asynchronous systems.

4.1.5 Procedural Code

Last but not least, the Java platform is not tied to the BotKit framework, FSMs, or any other aspects of programming style. This allows users to safely ignore the BotKit framework and directly access the hardware using native methods if they wish. One objective of BotKit was not to lock users into a certain frame of reference but give them a working alternative, and using Java helps with that goal. Of course, one would hope that everyone will like using BotKit, but that is partly a subjective as well as objective issue and will vary depending on individual preferences. It is important to note that the using Java to implement BotKit specifically still allows users to go their own way should they choose to do so.

4.2 BotKit

The BotKit framework is a collection of Java classes all under the BotKit.* hierarchy. The framework can be roughly broken into three major pieces: `AFSM` classes implement the state machines and also encapsulate procedural code within FSM wrappers, `Register` classes buffer inputs and outputs to AFSMs and outside sensors/actuators, and the `AFSMNetwork` collection binds everything together into one coherent mass and maintains global state amongst Registers, AFSMs, and encapsulated procedures.

4.2.1 Overview

BotKit was conceived as a software analogue to the hardware-based network used to implement the first subsumption architectures. It thus revolves around the concept of a network of finite state machines connected through register networks that convey information from AFSMs to inputs, outputs, and each other. The AFSMs themselves execute independently of each other in separate threads.

AFSMState

AFSMs consist of a collection of states and transitions between them (Figure 4.2.1). The states are themselves described by three class-specific methods: `guard()`, `enter()`, and `leave()`.

```
public class State {  
  
    /* guard method. must return true before state will be transitioned  
       into */  
    boolean guard() { ... };  
  
    /* executed once upon entering the state */  
    void enter() { ... };  
  
    /* executed once upon leaving the state */  
    void leave() { ... };  
}
```

Figure 4-1: The AFSMState class.

The `guard` method defined for each state sets out what conditions must be met in order to enter this state. Each time it is called, it should return `true` if the state's acceptance conditions are met, and `false` otherwise. The main FSM loop (implemented in the parent class) continually checks the `guard` methods for every state in the system to determine where to transition to next. By defining the entry barrier to each state as a function, flexibility is granted to the user and the state diagram can be constructed on the fly. A developer does not need to sit down and enumerate the entire state table beforehand because he or she can put down `guard` methods for the states that are known, and if turns out that staying in a particular state is not a good representation of reality then more states with their own access methods can be defined. If during the course of execution, more than one state's `guard` function evaluates to `true`, the main FSM loop will randomly pick one to execute. This allows for nondeterministic FSM behavior to be expressed. Of course, this is not a desirable trait in most systems, but that sort of model checking is best

left to other tools outside of the BotKit framework. Some of the future work detailed in Chapter 5 explores the idea of checking FSM models for completeness.

It is also common for states to have identical `enter` methods but only execute based on the previous state. For example, a robot that is backing up should execute turns to avoid obstacles that are opposite directions of the turns made while moving forward. At first glance, this seems hard to implement because the entry conditions for a state do not explicitly include what the previous state was. However, implementing `enter` as a method proves to be useful here. FSM classes maintain a private variable during execution that points to the currently active state. The `enter` method for a state can then be easily extended to check what the current state is in and add that to its list of conditions that must be satisfied before returning `true`.

`enter` is executed once when the AFSM transitions into this state. Typically, this method should be used to have the state perform actions associated with being in the state. For example, the state associated with a robot being left of a line being followed should take action to turn right towards the line here.

The `leave` method is executed once when the AFSM transitions out of that state into another. It allows users to have states perform cleanup duties that might be required.

AFSM

An AFSM in BotKit is defined to be a collection of states and the registers they draw information from. AFSMs implement the Java `Runnable` interface, which allows them to run as independent threads within the Java Virtual Machine. The main loop for an AFSM continually checks the guard conditions for all states in the FSM and determines what state the FSM should transition to next. Upon finding out what state to transition to, it executes the `leave` method of the current state and the `enter` method of the new state. An AFSM also has a default reset state which dictates what state the FSM starts out in upon activation.

AFSMs are not supposed to directly manipulate outside sensors and actuators; all communication outside of the AFSM goes through Registers. This rule is not

```

public class AFSM extends Runnable {
    Register[] registers;
    State[] states[];
    State resetState, currentState;

    void addRegister(Register r) { ... };
    void addState(State s) { ... };
    void setInitialState(State s) {
        resetState = s;
    }
    void reset() {
        currentState.leave();
        currentState = resetState;
    }
    void run() {
        /* loop through all states, call guard() for each.
           determine which state(s) should be transitioned to next. If no
           states qualify, stay in current state. If the current state's
           guard() returns true again, make a looped transition back to
           the current state.

           If a transition is made, call leave() method of current
           state, then set currentState reference to new state.
           Also call enter() method of the new state.
        */
    }
}

```

Figure 4-2: The AFSM class.

strictly enforced by BotKit, but the framework does not take responsibility for code that deliberately modifies hardware directly. The reasons for this suggestion are twofold: first, buffering inputs and outputs through registers makes preprocessing and postprocessing of data easier by allowing outside code to inspect and modify data in situ, and second, the register network allows the BotKit framework to avoid several potentially nasty problems that arise from multithreaded code.

Registers

BotKit Registers are responsible for managing the flow of data between AFSMs and the outside world. They are modeled closely after the hardware registers used to implement the first subsumption architectures in Brooks' robots. All BotKit registers carry integer messages; future iterations may carry more complex data types, but for now integers serve well. Also, certain special instantiations of Registers are responsible for reading data from outside sensors and/or sending data outside of the system to actuators and motors.

The most notable trait of a Register is its ability to integrate and combine inputs from multiple sources to produce one output value. This makes BotKit Registers more capable than their hardware counterparts. The default scheme is to take the latest, highest-priority write, but users may override this behavior if they wish.

Registers also employ an atomic update scheme that only allows newly-written values to overwrite the latched value when the global clock increments. This is essential to having all registers appear to update on a regular timescale, as explained in the next section. This scheme works by having the register keep a record of the last latch time in `clock`. When the global clock is greater than the current clock value, the register will latch the pending write value and update accordingly. If there have been no writes since the last update cycle by the time the clock updates, it just continues using the old value. There are special cases for when the register is first initialized and such, but those are left out for clarity.

This scheme allows reads to take place while writes are occurring to the register before the global clock updates. This way, it avoids the need for costly global update procedures and defers computation of the updated value until the register is actually read.

AFSM Networks

Systems implemented using BotKit must first inherit from the `AFSMNetwork` class. This class (Figure 4-4) is responsible for controlling overall interactions between AF-

SMS and their connected Registers. The central part of a AFSMNetwork instantiation is the `run()` method - this function is responsible for running the system's main loop. This loop continuously updates all registers and propagates their data throughout the system to the resident AFSMs can read it.

The original AFSM/subsumption implementations by Brooks used a hardware network of small microprocessors linked by wires. Since BotKit is a software implementation, a decision was made to emulate this network using multiple threads, one for each AFSM in the system, and buffered registers to handle communications between the threads and outside inputs/outputs.

In a hardware system with multiple functional units, registers are often used to buffer connections between modules. These registers exist to account for differences in propagation delay between modules; a piece of logic connected to another segment can put the system into a metastable state if it rapidly oscillates between different values and confuses a slower piece of hardware. These register networks are globally clocked such that the clock period is longer than the longest propagation delay for any one particular functional unit. This ensures that all logic has a chance to complete execution on one set of input values and present that output to the next stage.

If all the modules in a system have more or less similar execution times, this model works very well and balances a tradeoff between total computation latency and complexity in implementation. A completely asynchronous clocked system can in theory outperform a globally clocked system by operating all modules at peak speed, but in practice, ensuring the correctness of such a network is very difficult. However, if the modules in a system have widely varying propagation times, the time wasted by faster modules while waiting for slower modules to finish can grow to be quite considerable.

The same problem exists when taking this model to a software implementation. The AFSMs in BotKit by all execute at different speeds because they are not externally clocked. This design decision was purposefully made to allow time-sensitive FSMs to run as fast as possible while letting slower AFSMs for tasks such as pathfinding continue to operate at slower speeds. This makes a AFSMNetwork an asyn-

chronous system.

BotKit attempts to partially solve this problem by forcing all registers in the system to update according to a global clock. The `run` method in the `AFSMNetwork` class is responsible for starting all FSMs and updating registers continuously. Registers are updated simultaneously through use of a global clock that registers watch. While the clock is still, writes to the register do not actually modify the latched value, but once it updates, the registers will update on the next call to `read` as previously described.

Having all the registers update in sync with each other and the clock helps avoid most classes of problems arising from multithreaded code. However, it does not completely eliminate all of them, because the ASFMs reading the registers may still end up reading registers just before they get updated and use those values for calculations that are no longer valid. At this time, there is no easy way to solve this problem short of putting the whole system on a global clock, but if some attention is paid to constructing the ASFMs to execute quickly response times can still be kept to an acceptable maximum.

4.2.2 Example Use

Line Follower FSM

As an example, let us try to implement the line-follower behavior using BotKit. The code for the FSM itself looks like the skeleton in Figure 4-5.

Note that the various actions, guards, and transitions for the state `leftOfLine` are implemented using Java anonymous local class definitions. Anonymous local class definitions allow users to define the methods of a class at the instance of creation without having to predefine all the code before instantiation.

The only code executed for a state occurs at the beginning and end of the transition. A state may choose to transition back to itself if it wants actions to repetitively execute; to do that, the `guard()` method should return true as long as the conditions required exist. Otherwise, the `guard()` method should make sure that the FSM is

not already in the target state if the `enter()` method is only to be run once.

The code inside AFSMs can be used to do anything; for the purposes of the contest, users were taught how to use the class hierarchy and embed their robot code inside FSMs. However, the BotKit AFSM model is not tied to the Robot Controller class library and can be used to model other systems not related to robotics.

Line-Follower Procedure

The mechanism for encapsulating procedural code in BotKit along with FSMs follow much the same sequence as that for a real FSM. In the current implementation, encapsulation is accomplished by wrapping a procedure or collection of procedures inside a Java class that looks the same as a ASFMS externally. It has wrapper functions to expose register contents and manipulate output registers internally, so all connections to this “pseudo-FSM” appear just as if it is another AFSM in the network (see Figure 4-6). This fulfills the promise of allowing procedural code to coexist with state machines in the subsumption architecture.

This `LineFollower` AFSM runs in its own thread of execution, and the sensors are updated with each call to the `read()` method. This ensures that sensor reads are only performed when the data is actually needed. For sensors that are read more efficiently at regular intervals, the `Sensor` class can be overridden to only sample new data at certain times and cache the data returned by `read()`.

4.2.3 Line Follower + Distance Measurer

Of course, having an AFSM is not much better than running standard procedural code unless the behavior is combined with another AFSM. In this example, another AFSM that makes the robot only travel a certain distance before stopping is layered on top of the line follower. Note that the code in this AFSM is not very complex – it simply measures how far the wheels have turned and stops the motors if the wheels have made a certain number of revolutions. For such a simple task, a procedure implements the code (Figure 4-7) much more compactly than a FSM can, so this

AFSM is implemented using an embedded procedure instead of a FSM.

Combining these two behaviors is done through setup routines that get called before the system is initialized. The registers are connected to each other and other AFSMs here. The inhibitory/combining relationships between relationships are also set up, and this is what allows AFSMs to combine behaviors without having explicit communication between each other. The code in Figure 4-8) illustrates these operations.

4.3 Implementation Notes

I worked in conjunction with a team of researchers and engineers from the Compaq Cambridge Research Laboratory to finish Robot Controller and get its software working. The hardware design was a joint effort between the research staff and CRL's Advanced Systems Engineering (ASE) group. I had input into the designs and specifications of the Personal Server, and supplied most of the design parameters for RoboDC due to my 6.270 experience.

The hardware debug and software work was done by several people at CRL, including myself. I was heavily involved in getting the first RoboDC daughtercards working and wrote the first pass of the firmware, and also wrote the first bootloader for the Personal Server motherboard. The NetBSD and Kaffe JVM ports were also areas that I substantially contributed in. CRL helped complete the effort by making contributions in all those areas and provided valuable expertise in designing the USB protocols and C API libraries used to communicate between PersonalServer and RoboDC.

Unfortunately, the actual port of the BotKit framework to the Robot Controller controller was not completed in time for the 6.270 competition because so much effort went into making the controller itself work. However, we were able to supply a base Java development environment and Robot Controller class libraries to abstract the hardware away from users. We did also provide some helper classes to encapsulate Behaviors and at least provide some introduction to subsumption. These Behavior

classes, coupled with helper Timer classes that would go off after a certain amount of time, were deployed in time for the contest.

4.4 Time Frame

The time frame for the whole project spanned approximately nine months. Three months were spent working on the BotKit framework and Java AFSM/encapsulation classes during the Summer of 1998, and a software-only demonstration was given at the end of August 1998. In August, hardware specification and design commenced on the Robot Controller controller, and the first prototypes came back from the fabrication facility in November 1998.

Most low-level software work was done between November 1998 and January 1999. Due to the hard deadline involved, all work was stopped on the BotKit AFSM framework and resources dedicated to the hardware/software debug on Robot Controller. The NetBSD kernel first booted sometime in the beginning December, and Robot Controller executed its first Java code on December 28, 1999. The first complete Robot Controller system capable of driving motors and sampling sensors from Java code was brought to life late during the night of January 3, 1999.

After much work, 55 Robot Controller systems were delivered to the MIT 6.270 contest on January 8, 1999.

```

public class Register {
    int pendingWriteValue; /* incoming writes */
    int pendingWritePriority; /* incoming write priority */
    int clock;

    int latchedValue;

    public synchronized void write(int value, int priority) {
        /* examine the incoming value/priority pair and replace
           pendingWriteValue/pendingWritePriority only if the priority is equal
           to or higher than the current priority. */
        if (AFSMNetwork.clock > clock) update();
        if (priority >= pendingWritePriority) {
            pendingWriteValue = value;
            pendingWritePriority = priority;
        }
    }

    public synchronized int read() {
        if (AFSMNetwork.clock > clock) update();
        return latchedValue;
    }

    private void update() {
        latchedValue = pendingWriteValue;
        pendingWritePriority = 0;
        clock = AFSMNetwork.clock;
    }
}
}

```

Figure 4-3: The AFSMRegister class.

```

public class AFSMNetwork {
    public int clock;
    AFSM[] fsms;
    Register[] regs;

    void addAFSM(AFSM fsm) { ... };

    void run() {
        /* start all AFSMs */
        clock = 0;

        while (true) {
            /* update clock at set interval */
            /* keep track of error conditions, stuck FSMs, etc. */
        }
    }
}

```

Figure 4-4: The AFSMNetwork class definition

```

class LineFollower extends AFSM {
    Register leftSensor, rightSensor, middleSensor;
    Register leftMotor, rightMotor;
    State leftOfLine, onLine, rightOfLine,
        offOfLine, onJunction, slantJunction;
    LineFollower(Register ls, Register rs, Register ms) {
        leftSensor = ls;
        leftOfLine = new State() {
            public boolean guard() {
                /* executed every time FSM checks what state to
                   transition to next */
                return (!leftSensor.read() && !middleSensor.read() && rightSensor);
            }
            public void enter() {
                /* executed so long as FSM remains in this state --
                   run the left motor faster than the right motor to turn right.
                */
                leftMotor.write(100,1); rightMotor.write(70,1);
            }
            public void leave() {
                /* executed before FSM switches to a new state */
            }
        }
        [ etc.. repeated for other states ]
        /* set up the FSM and add registers/states to the table */
        addRegister(leftSensor); addRegister(rightSensor);
        addRegister(middleSensor); addRegister(leftMotor);
        addRegister(rightMotor);
        addState(leftOfLine); addState(onLine); addState(rightOfLine);
        addState(offOfLine); addState(onJunction); addState(slantJunction);
        /* set the start state - the FSM will begin execution in this
           state */
        setInitialState(onLine);
        /* start the main loop running. this loop will keep checking
           enter() for each state and transition as necessary. */
        run();
    }
};

```

Figure 4-5: The line follower as an AFSM

```

class LineFollower extends AFSM {
    Register leftSensor, rightSensor, middleSensor;
    Register leftMotor, rightMotor;
    LineFollower(Register ls, Register rs, Register ms,
                 Register lm, Register rm) {
        leftSensor = ls; rightSensor = rs; middleSensor = ms;
        leftMotor = lm, rightMotor = rm;
    }
    void drive(byte left,byte right) {
        leftMotor.write(left,1);
        rightMotor.write(right,1);
    }
    void run() {
        while (true) {
            /* do processing here procedurally */
            if(middleSensor.read() && !rightSensor.read()
               && !leftSensor.read()) {
                /* On Line */
                drive(100, 100);
            } else if(leftSensor.read()) {
                /* Turned to the right.
                 Turn back left to correct */
                drive(20, 80);
            } else if(rightSensor.read()) {
                /* Turned to the left.
                 Turn back right to correct */
                drive(80, 20);
            } else {
                /* None of the sensors read a line.
                 Do nothing - there is no line to follow */
                continue;
            }
        }
    }
};

```

Figure 4-6: The line follower as a procedure encapsulated in an AFSM


```

class DistanceMeasurer extends AFSM {
    Register leftEncoder, rightEncoder;
    Register leftMotor, rightMotor;
    int counter;

    DistanceMeasurer(Register le, Register re,
                    Register lm, Register rm);
        leftEncoder = le; rightEncoder = re;
        leftMotor = lm; rightMotor = rm;
        counter = 0;
    }
    void run() {
        int distanceTraveled;
        while (true) {
            counter = (leftEncoder.read() + rightEncoder.read())/2;
            if (counter >= distance) {
                leftMotor.write(0,2);
                rightMotor.write(0,2);
                signal.write(1,1);
            } else {
                /* this is an error condition. in a more fleshed
                out implementation, this should write raise some
                sort of error signal */
            }
        }
    }
};

```

Figure 4-7: Distance Measurer AFSM Implementation

```

public class robot extends AFSMNetwork {
    static Register leftSensor,middleSensor, rightSensor,
                    leftEncoder, rightEncoder;
    static Register leftMotor, rightMotor;
    static DistanceMeasurer afsmDM;
    static LineFollower afsmLF;

    public void main() {
        leftSensor = new Register(new RoboSkiff.AnalogSensor(1));
        middleSensor = new Register(new RoboSkiff.AnalogSensor(2));
        [...]
        rightEncoder = new Register(new RoboSkiff.AnalogSensor(5));

        afsmDM = new DistanceMeasurer(leftEncoder,rightEncoder,
                                     leftMotor,rightMotor);
        afsmLF = new LineFollower(leftSensor,middleSensor,rightSensor,
                                 leftMotor, rightMotor);

        addAFSM(afsmDM);
        addAFSM(afsmLF);

        run();
    }
}

```

Figure 4-8: Example robot combining Line Follower and Distance Measurer behaviors

Chapter 5

Results and Analysis

The month of January went by very quickly. Fortunately, the RoboSkiff boards actually performed fairly well, except for some glitches with the hardware. However, the software itself performed very well, and with some small exceptions, most users were able to produce working robots with it.

Due to problems delivering the initial software images and power cables for the controllers, 6.270 contestants were unable to start booting the controllers until the second week of the class. However, the first week was primarily spent working with LEGO® and teaching students how to build mechanical structures, so the lost time was not missed quite so badly. When the boot images did get released, all the boards worked, which was a relief to the development team, and to me in particular.

Throughout the month, daily updates were made to the development software to fix several small bugs and make changes to the class libraries. Constant feedback from the contestants helped make the development system much better as a result. We were fortunate in having a group of contestants who for the most part were excited about getting the chance to work with new hardware for the first time. Some contestants actually helped out tremendously by contributing code and hardware fixes back to the development team.

5.1 NetBSD/Kaffe combination

The base NetBSD operating system and Kaffe Java Virtual Machine performed very well throughout the contest. No known errors or faults with either component were reported. There was some initial confusion because the Kaffe JVM is not JDK 1.2 compliant, but an announcement to the contestant mailing list solved that issue very quickly. All in all, using NetBSD on RoboSkiff proved to be one of the better decisions the development team made.

The USB support under NetBSD proved to be quite robust and was able to handle some rather adverse conditions, usually involving failure of RoboDC daughtercards due to motor overloading. To the best of our knowledge, no USB-related errors were reported throughout the contest.

5.2 Java

One major benefit of the Java development environment used for RoboSkiff showed up immediately when a small group of contestants asked about Macintosh support. We had not planned for this possibility, and under the old system it would be been out of the question to attempt supporting a new platform during the contest. However, a download of the Apple Java Virtual Machine and Java Development Kit to the Macintosh proved to work very well, and all RoboSkiff code compiled fine for RoboSkiff on the Mac without a hitch. The same scenario was repeated many times on other platforms like Linux, NetBSD, and even one AmigaOS user¹. The platform-independence of the Java platform really helped our efforts in making RoboSkiff development accessible to everyone.

The learning curve inherent to Java did cause problems for some people that were used to C, but contestants new to programming in general actually found Java to be reasonably accessible. I managed to publish quite a bit of example code throughout

¹Note: After that experience, I can definitely say that running the Java VM on an Amiga is not for the impatient.

the contest to show how to perform basic tasks within the RoboSkiff and Behavior frameworks, and many users found those examples to be helpful.

Overall, the responses of contestants after the contest generally indicated that Java was a good thing. There were obviously some people who just found C to be more natural due to their years of experience using it, but the cleaner syntax and object-oriented features won quite a few contestants over to the Java camp. The thread support also helped quite a bit for those teams implementing very complex control systems. Incidentally, those were the teams who also used the Behavior classes to good effect.

The general conclusion on the Java decision is that while it was a good one, more work can definitely be done to lower the initial learning curve and also support higher-end users. Better design of the class libraries and perhaps distribution of 3rd party development tools with real Integrated Development Environments would also help as well.

5.3 BotKit

Contest organizers are responsible for teaching recitations during the month of January. These recitations, with about eight to ten teams each, are meant to serve as a place for further instruction and question-answer discussion regarding the progress of teams' robots throughout the month. To test whether subsumption theory would make an impact on contestants, I led two recitations with a total of nineteen teams throughout the month and taught them what I knew about the subsumption architecture and FSM modeling of robot systems. In addition to teaching students how to model systems using state machines, I also attempted to give examples of how to use the Behavior classes to full advantage and use multithreading to layer behaviors on top of each other. The other organizers generally taught standard procedural techniques.

Three out of the top five teams in the contest came from my two recitations (there were six recitations total). All three of these teams used state machines to model their

robot behaviors, and they exhibited some very interesting behavior in the final rounds against several robots with pre-planned strategies. This also illustrates an interesting side-effect of the winners – the two robots who rounded out the top five won due to mechanical advantages and not software reliability. Overall, twelve of the teams in the contest used FSMs in some form (ten from my recitations, and two from other recitations) for their final contest entries. However, only three teams of those twelve actually modeled their robot using subsumption. Most teams chose to instead use FSMs for modeling parts of their robot, like line navigation, and relied on procedural code to manage top-level behavior by starting and stopping FSMs as necessary. However, the teams that used FSMs generally reported good results with them provided the FSMs were specified correctly. Teams with limited programming experience in particular found FSM modeling to be simpler than writing nested `while/if` loops to manage feedback loops, proving the validity of this approach. Unfortunately, there was rarely more than one FSM running at a time, which limited the robots to doing roughly one thing at a time sequentially. The intricacies of multithreaded programming proved to be too difficult for most teams to master without the register abstractions provided by a full BotKit implementation.

The main argument heard against using only subsumption for all robot behavior was that it seemed hard to write goal-oriented behavior layers. The subsumption ideas held very true for repetitive feedback systems attempting to maintain a certain stable state, but when series of goals were set down the layered approach proved impractical. This illustrates the need for procedural encapsulation frameworks like BotKit that allow both types of code to be used in the same system. One contestant put this predicament fairly succinctly when he termed subsumption as a good idea for “don’t worry, be happy” robots – robots that can wander aimlessly around a room and stay out of trouble, but do not do anything useful. Of course, with some work, a subsumption-only approach can definitely be made to perform goal-based tasks, but students with limited experience definitely preferred the procedural approach.

The AFSM concepts were not accepted by everybody. Many teams attempted to implement state machine networks in their robots, but had to give up after difficulties

arose in breaking down systems into states. Without the BotKit framework to encapsulate procedural code within the networks, many teams gave up and started over using purely procedural methods. The general opinion coming out of the contestants in my classes was that state machines sounded extremely useful in recitations, many teams just were not able to understand how to go about splitting systems into states. The procedural view of the world is still very strong.

5.4 Future Work

Since the entire BotKit framework was not deployed this year, a high-priority goal for next year is to improve it and deploy it under contest conditions next year. The hardware platform for next year should be much more stable and so more time can be devoted to actually working on the framework and improving it. Additionally, many other areas remain open for research that might substantially improve the usefulness of BotKit as a rapid prototyping tool for roboticists in the MIT 6.270 contest as well as embedded system developers in general.

5.4.1 FSM description tools/languages

Networks of FSMs provide a powerful means to express and implement an embedded system. However, there is still the issue of how to go about expressing these networks in machine-executable form once they are designed. Many different approaches to performing this task have been proposed, and active research is still ongoing in this field. Some approaches have focused on making the act of constructing FSMs easy through graphical tools or specialized programming languages. The Argos system is one such example of graphical FSM design tools[16].

Several languages for specifying and compiling state machine descriptions to either lower-level languages like C or even hardware descriptions also exist. One example in somewhat widespread use is Esterel[9]. Esterel is actually more than just a language for expressing FSMs - it encompasses an entire framework for specifying and implementing synchronous reactive systems. The compiler is capable of generating both

software and hardware implementations of finite state machine systems, optimizing, and other useful things.

However, Esterel's language syntax is significantly different from mainstream languages. As previously stated, this poses problems to the target audience for BotKit because they do not have the time to learn a new language from scratch. BotKit's goal was to implement the subsumption/AFSM framework in a programming language readily accessible to a large number of people.

5.4.2 FSM Verification

Some teams in the contest ran into problems with under-specified FSMs - they had too few states to express all possible input combinations. While the systems did not crash (as probably would happen in a procedural system), these problems proved to be very difficult to find and fix. Much research has been put into the area of FSM verification and simulation[14], and future work might include borrowing from that research to help users construct robust AFSMs in BotKit

5.4.3 NetBSD/JVM work

The underlying NetBSD operating system for Skiff probably will not change beyond updates to the mainline source tree, but some directions for the Kaffe JVM look promising. In particular, this year's release of the RobotController software did nothing to take advantage of Java's dynamic code loading ability. This made the RoboSkiff development environment somewhat slower than Interactive-C because compile steps were needed just to try out simple tasks. In the future, a Java "shell" for the controller that is capable of loading compiled classes and directly calling methods would go a long way towards making "Interactive-Java".

5.5 Conclusion

The idea of encapsulating procedural code along with AFSMs in networks shows promise, and preliminary results from the MIT 6.270 contest have shown that it is possible to increase productivity and encourage more developers to write robust code using FSMs. Much work remains to be done, but future 6.270 organizers will hopefully pick up and continue teaching reactive control methods to future contestants.

BotKit marks an attempt to implement Brook's subsumption architecture in the Java programming language, and enough work has been done to show that this combination can be useful even to developers with limited experience in either. Previously, the subsumption ideas have been implemented using Lisp-like languages on networks of multiple processors[5], and BotKit moves away from both of those norms. We have shown that a subsumption architecture can be implemented in a mainstream programming language. In addition, procedural code can be encapsulated and used within that architecture, allowing users, especially beginners, to gradually adjust to the FSM model and combine small specialized code fragments in a robust manner.

Bibliography

- [1] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. In *Proceedings of the IEEE*, 1991.
- [2] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, pages 14–23, March 1986.
- [3] R. A. Brooks. A hardware retargetable distributed layered architecture for mobile robot control. In *Proceedings of 1987 International Conference on Robotics and Automation*, pages 106–110, 1987.
- [4] R. A. Brooks. A robot that walks; emergent behaviors from a carefully evolved network. In *IEEE International Conference on Robotics and Automation*, pages 292–296, 1989.
- [5] R. A. Brooks. The behavior language user’s guide. MIT AI Lab Memo 1227, April 1990.
- [6] R. A. Brooks. Integrated systems based on behaviors. *SIGART Bulletin*, pages 46–50, August 1991.
- [7] R. A. Brooks. New approaches to robotics. *Science*, pages 1227–1232, September 1991.
- [8] S. A. Edwards. *The Specification and Execution of Heterogeneous Synchronous Reactive Systems*. PhD thesis, University of California, Berkeley, 1997.

- [9] Georges Gonthier Gerard Berry. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2), 1992.
- [10] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
- [11] J. Jones and A. Flynn. *Mobile Robots: Inspiration to Implementation*. A. K. Peters Ltd., Wellesley, first edition, 1993.
- [12] A. Girault B. Lee and E. Lee. Interaction of finite state machines with concurrency models. In *Proc. of 32nd Annual Asilomar Conference on Signals, Systems, and Computers*, November 1998.
- [13] Alain Girault Bilung Lee and Edward A. Lee. Hierarchical finite state machines with multiple concurrency models. Technical Memorandum UCB/ERL M97/57, October 1998.
- [14] S. Edwards L. Lavagno E. A. Lee and A. Sangiovanni-Vincentelli. Design of embedded systems: Formal models, validation, and synthesis. In *Proceedings of the IEEE*, 1997.
- [15] P. Maes and R. A. Brooks. Learning to coordinate behaviors. *AAAI*, pages 796–802, August 1990.
- [16] F. Maraninchi. The argos language: Graphical representation of automata and description of reactive systems. In *Proceedings of the IEEE Workshop on Visual Languages, Kobe, Japan*, October 1991. <ftp://ftp.imag.fr/pub/SPECTRE/ARGONAUTE/ArgosIEEEVisual.ps.gz>.
- [17] F. Martin. *Circuits to Control: Learning Engineering by Designing LEGO Robots*. PhD thesis, MIT, 1994.
- [18] M. McKusick K. Bostic M. Karels J. Quarterman. The design and implementation of the 4.4bsd operating system, 1996.

[19] Unknown. An introduction to 6.270 and its history.
<http://web.mit.edu/6.270/www/intro.html>.

Biographical Note

Edwin Foo was born on April 11, 1979 in Malaysia and emigrated with his family to the United States in 1985 after spending parts of his childhood in both Malaysia and Singapore. He graduated from Clear Lake High School in Houston, Texas in 1994. On June 4, 1999, Edwin will receive simultaneous Bachelors and Masters degrees in Computer Science with a concentration in Systems and Architecture. After graduation he will be working as a member of the Technical Staff at the Compaq Cambridge Research Laboratory. He is still an active organizer of the MIT 6.270 Autonomous Robot Design Competition and keeps vast amounts of LEGO® in his office at Compaq CRL in order to attract visitors and the children of fellow researchers (sometimes their parents as well).