A Laboratory Course in Behavior-Based Robotics

Ian Horswill
Computer Science Department
Northwestern University
1890 Maple Avenue
Evanston, IL 60201
ian@cs.northwestern.edu

Abstract

The Behavior-Based Robotics course at Northwestern University is a project-oriented course that gives undergraduate and graduate students exposure to programming research-grade robots for real-time autonomous activity. The course, which has been running for five years, combines lectures on theory with large amounts of concrete project work in which students are encouraged to creatively select their own goals and approaches. One of the most unusual aspects of the course is its use of low cost, state-of-the-art hardware that provides not only SONAR and odometric sensing, but also a number of real-time vision systems. We will discuss the curriculum of the course and our experiences with it. We will also discuss the evolution of the hardware and software infrastructure used in the course and its impact on the students' learning experience.

Keywords: Robotics, Behavior-based systems, Education.

Submitted to IEEE Intelligent Systems

Introduction

Robotics has received a great deal of attention as a vehicle for motivating students to learn artificial intelligence, computer science, and even general science and engineering [2]. A large number of courses and competitions have been developed that involve the construction and programming of simple robots. These robots are usually composed of a microcontroller, some bump switches and IR receiver-detector pairs, motors, and a body built using a mechanical prototyping technology such as Lego Technic. While these courses are excellent vehicles for teaching general engineering concepts, especially electrical and mechanical engineering, they tend to be dominated by hardware concerns. The kinds of issues that robotics researchers think of as being core concerns in designing autonomous robots – representation, planning, sensor fusion, etc. – can't be addressed because the robots lack the sensor suites needed to make these problems practical.

At Northwestern, we have taught a laboratory course for the last five years that uses research-grade mobile robots to explore concepts of autonomy, such as sensor interpretation, planning, and action selection. In this paper, we will discuss the history of the course - the different approaches we tried, the pitfalls we encountered, and the successes we encountered.

Goals

The principal goal of the course was to teach students the issues and approaches involved in modern autonomous mobile robot programming. This was partly a selfish goal. Autonomous robots require large amounts of mundane infrastructure, from device drivers to simple vision algorithms, that, while necessary, is not in itself publishable research. Most autonomous robot labs are therefore dependent on highly motivated undergraduates for whom such tasks are still fresh and exciting. Teaching a robot course that provides background on current research topics is both a way of recruiting these students and a mechanism for training them in.

We wanted to teach a laboratory course rather than a straight lecture course. This was due in part to a general bias toward project-oriented courses within our department. We believe that students learn better by doing than by sitting through lectures or doing pencil and paper exercises. However, the lecture component of the course has grown over the years and there is now a large amount of lecture material in addition to the laboratory assignments.

Our eventual goal is to teach a course that starts with linear control theory and moves all the way through symbolic planning and knowledge representation. For various infrastructure reasons (see below), that hasn't yet been practical. We have therefore focused the course on behavior-based control of visually guided robots. When we teach the course this fall, we intend add units on sequencers and STRIPS planning.

Course lectures

The course uses Arkin's *Behavior-Based Robotics* [1] as its primary text. Since the book covers the material at a relatively high level, we supplement it with lecture notes on the nuts and bolts of robot programming. There is also a large amount of documentation on the programming environment and utilities. Although the course is not ultimately about traditional control theory, we have organized the course around the vocabulary of control theory:

The environment and robot form **coupled dynamic systems**.
Dynamic systems have **state spaces** and **dynamics** (transition functions or differentials)
State estimators compute an approximate state for the robot/environment from old state estimates, the actions the robot has taken, and new sensor data.
Control policies compute actions from estimated states and/or sensor data.

The course begins with a brief review of dynamic systems where these ideas are presented for both discrete and continuous state spaces. We then discuss rudimentary control theory (for linear single-input/single-output systems) as a starting point for thinking about robot programming. Unfortunately, typical commercial mobile robots are controlled by sending velocity commands over a low-speed serial port. To a control theoretician they look like first-order systems with long lags and peculiar nonlinearities. This makes it difficult to design lab activities in which students can apply the formal aspects of classical control theory. If we ignore the lags and non-linearities, robot bases look like trivial systems that scarcely need to be controlled. However, if we include the lags and non-linearities, they look too complicated to model in an undergraduate course, much less one that only can spend a week or so on control theory. For this reason, we focus on the qualitative aspects of classical control, such as how lags induce oscillation, or how stiction can induce integrator wind-up by breaking a feedback loop. We also discuss the typical work-arounds for these problems, such as controller detuning, *ad-hoc* gain scheduling, and conditional integration. Ultimately, we would like to extend the course to cover adaptive control. However, this will probably require extending the course to a two-quarter sequence.

Most of the rest of the course is devoted to behavior-based control and behavior-based architectures. We begin by introducing the notion of *primitive behaviors* as gated control loops. These are task-specific controllers (typically non-linear) together with some kind of triggering logic. The triggers enable their control loops when the robot/environment system enters the appropriate region of its state space. Control policies are then built by combining primitive behaviors. Different techniques can be used to combine behaviors and we can think of these techniques "higher-order" operators that map behaviors to more complex behaviors. We provide the students with implementations of a number of these combination operators, or *combinators*, allow them to experiment with them, and allow them to try developing new combinators.

For a behavior-based control policy to be effective, it must activate the right behavior at the right time. More properly, it must activate the right behavior in the right region(s) of state space. Standard computer programs use sequencing as their major way of combining primitives. First you call this routine, then you call that routine. Unfortunately, sequential controllers, also called *plans*, require position within the plan to correspond reliably with location in state space in order to function properly. Thus, the plan's program counter must effectively encode some kind of *ad-hoc* state estimate. While this is perfectly reasonable in a sort routine, it can be difficult to achieve reliably in robot programs if the environment is stochastic or poorly modeled.

For this reason, behavior-based controllers generally combine behaviors in parallel. The triggers act as specialized task-specific state estimators and the system runs whichever behavior(s) are triggered at a given time. Such parallel control policies are effective when (roughly):

The behaviors do sensible things in their respective regions of the state space,
Their triggers accurately detect these regions of state space, and
Those regions cover the state space.

In practice, of course, the regions overlap, meaning that multiple behaviors can be coactive, so some kind of arbitration is necessary. Different behavior-based architectures break down roughly according to what arbitration mechanism they use. Motor schemas use vector summation. Subsumption uses a prioritization scheme. Behavior-nets use spreading activation. And so on. In laboratory exercises, students are given implementations of each of these combinators and told to experiment with them on different tasks. Coming from a computer science background, our students generally have a bias toward using sequential control. Their first few robot programs tend to be filled with large numbers of ad-hoc timers and sequencers that are carefully contrived to cause the robot to do the right thing in the imagined "typical" case, but that tend to fail if someone unexpected walks in front of the robot or even just if shadow moves in the environment. Much of the focus of the first few assignments is to wean them from sequential control.

Issues of hardware design, sensing and state estimation a distributed throughout the course. The course begins with a discussion of common robot hardware: batteries, power-train components, infrared proximity detectors, SONARS, cameras, computers, etc. These topics are discussed at the functional level of what they're supposed to do and how they commonly fail, rather than at the level of how a student would design a new robot.

A fair amount of class time is devoted to the sensor failure modes, particularly specular reflection and corner reflection of SONAR pulses. The sources of drift in odometric position estimates are also discussed at length. Although these issues are discussed at the beginning of the quarter, students are always shocked and outraged when their robots crash into the wall because of a "bad" SONAR.

We also discuss simple active vision algorithms. The students do most of their navigation using the Polly algorithm [3] for obstacle sensing, and so become experts in its failure modes. The algorithm can't sense any object that doesn't generate a grey-level edge in the image, so objects that happen to be the same brightness as the floor are invisible to it. It is not uncommon to come into the lab in the morning and find black electrical tape strung along the bottom of some recalcitrant piece of furniture.

Stochastic state estimation is covered toward the end of the course. For continuous state spaces, we discuss least squares fitting and recursive least squares. Kalman filtering is also discussed, although no real attempt is made to prove its correctness in class. For discrete state spaces, we discuss Bayesian inference and Markov models. While we explain the basic ideas behind Markov Decision Processes (MDPs) and Partially Observable MDPs, we do not attempt to present the literature on POMDP planning.

We also discuss hierarchical organizations, such as means-ends analysis and Tinbergen hierarchies. Next year, we hope to add a real unit on symbolic problem solving and planning. We believe we can integrate it cleanly with behavior-based curriculum by treating planning as another kind of combinator, but one which uses non-determinism (i.e. search), rather than parallelism. Non-deterministic programming languages been shown to be an accessible way of teaching search algorithms in introductory AI classes.

Lab assignments

Lab assignments are performed by the students in the hallways of the computer science department. Students are not allowed to modify the environment to suit their needs, nor are they allowed to ask passers by to move out of the way or to otherwise avoid the robots. In this sense, the entire building forms a kind of "open lab" for the students' experiments. This serves a number of purposes. To begin with, it makes the assignments much more realistic. Students coming from "Lego robotics" background, where they compete in specially designed, tightly controlled robot rinks, often don't understand why the techniques of behavior-based control are necessary, since simple timing loops and odometry often suffice in these artificial rinks.

However, a more important reason is that putting lab assignments out in the world helps generate a sense of excitement, not only among them, but also among other students. As students go about their day, they walk past robotics students frantically hacking away on their latest assignment. This obviously helps motivate the robotics students, but it also helps build a general sense among the other students that there are cool things happening in the department.

Students work on assignments in groups of 2-4. They are given 24-hour access to the robots. While this makes the course popular with the students, it does have some disadvantages. First, students tend to book the robots 24 hours a day in the final day or two before an assignment is due. That means that the teaching staff needs to closely monitor the condition of the robots to make sure that the batteries are not being overdischarged, that the wheels are still aligned, etc. Unfortunately, it also means that the teaching staff has to monitor the condition of the students, since they have a tendency to spend more time on their robotics assignments than is really healthy for them or for their grade point averages. Working through the night is the norm in the class. It is not uncommon for student to stay up 24 hours. In some cases, students have even gone for 48 hours or more. It is important for the teaching staff to monitor the students and encourage them to go home and sleep, eat, shower, and so on.

Most assignments are structured as semi-formal competitions. It's not clear to us that this is a good thing. However, experience has shown that the students will turn any assignment we give them into a road race whether we like it or not. The only way we have found to prevent them from trying to make the robots run as fast as possible is to give them some other criterion, such as energy consumption or smoothness of velocity, to try to optimize. This may be a reflection of the predominantly male undergraduate population in CS. In any case, we have reluctantly decided to embrace their enthusiasm by structuring assignments as mock-competitions. Each assignment specifies:

- A number of trials that each group will get and a duration for each trial.
- A set of formal benchmarks, such as time, velocity, or energy that will be used to compute a quantitative evaluation of the system.
- A set of constraints, such as requiring that robots not collide with any objects or that they never back up. These count as penalties if they are violated.
- A set of subjective criteria, such as perceived smoothness of the robots' trajectories, general coolness, etc.

The teams have one week to do the assignment, at the end of which trials are held in class. Each team gets a specified number of trials and statistics are recorded about their performance in each trial. The students are graded based on the statistics from their best trial. The students are also required to give a 5 minute in-class presentation on how their system works. Finally, the students are asked to grade *each other* on the subjective criteria.

Most assignments involve the design of a specific behavior, such as:

- *PD-controller for ballistic turns*. Groups compete for the fastest settling time. This is really just an assignment to get them used to working with the hardware. It has the advantage that the robot doesn't actually translate, so it isn't as bad when they forget to remove the charging cable before enabling the motors. While a simple task, it isn't entirely trivial as it sounds since the plant they're controlling has a number of non-linearities.

- *Freespace follower*. Students are given the basic visual operators for computing a depth map and for computing the minimal distance within regions of the map. The students then build a reactive control loop to wander about the lab avoiding obstacles. Here students begin to experiment with techniques like gain scheduling. Some years, this is given as a boundary follower instead of a freespace follower.
- *Wanderer*. Groups augment their freespace follower with an "unwedger", a behavior that detects when the freespace follower has reached a *cul de sac* or other local minimum and overrides it with a ballistic turn or other motion to get it out of the local minimum. The wanderer assignment comes in different flavors:
    - *Normal*. Groups are scored based on the maximal distance they get from their starting point in a set period of time.
    - *Limo mode*. The run-time system of the robot is instrumented to compute the RMS value of the robot's linear acceleration. Groups compete for the lowest mean acceleration. This encourages smooth velocity profiles, i.e. low energy trajectories.
    - *Energy saver*. An alternative form is to instrument the run-time system to compute the total energy expenditure through the motors based on measured battery voltage and motor currents. Students then compete for the highest mean velocity sustained using the smallest number of joules of energy over a standard period of time.
    - *Marathon man*. The run-time system of the robot is instrumented to compute the minimum velocity over the course of the robot's run. Groups compete to achieve the highest possible minimum velocity. This also encourages smooth, lower-energy trajectories.
- *Tailing*. Students write a behavior to track a person and follow them. Groups compete to design the most responsive behavior possible.

However, there are also assignments that involve building more complete “applications”, such as:

- *Road race*. This is a race from Ken Forbus' office in the East wing of the building to Roger Schank's office in the west wing. Students have to tune a freespace or boundary follower to run as fast as possible. They also typically add some kind of simple sequencer and landmark detector to switch modes when they reach different locations in the lab.
- *Fetch*. Given visual primitives for tracking a brightly colored ball, students write a behavior-based system to play fetch.
- *Place recognition*. Groups write a number of *ad-hoc* landmark detectors and build a topological navigation system based on these detectors. Groups compete for accuracy in detecting their chosen landmarks.
- *Town crier*. Groups write a program to patrol the lab and announce a talk or other event.
- *Pest*. Groups write a program to lurk in the hallways and accost passers by.

At the end of the quarter, the students do a final project of their own choosing. Some students build useful bits of infrastructure for the robots, such as a network server to allow students to control the robots remotely. Others try to do a better job on some past assignment that they really enjoyed. Many students try to apply ideas from other classes to the robots, such as interfacing a planner to an existing suite of robot behaviors.

Hardware

One of the most difficult aspects of the course has been that we have gone through four different generations of robot hardware in five years. Our requirements for the course hardware were:

- *Low cost*. In order to properly service a lab course, we need many copies of the hardware. This means the robots have to cost less than $10,000 and ideally should cost less than $2,000.
- *Durability*. The robots have to be able to withstand repeated crashes at 1 m/s. The batteries must be able to withstand accidental deep discharge without failing.
- *Safety*. The robots have to be able to crash into humans at 1 m/s without causing serious damage to either party. Sharp corners must be removed. Laser range finders must be eye safe.

- *Debugability*. It can be very difficult to distinguish between bad code and bad sensor data, even for experts. For students to have a fighting chance of debugging a complex multi-behavior system, there must be some way for them to obtain real-time displays of the robot's internal state.
- *Real-time vision*. Sensing modalities such as SONAR, bump switches, etc., do not naturally parse the world into objects. They simply report the presence and non-presence of undifferentiated matter. While this supports many tasks, such as collision avoidance, boundary following, and even landmark detection, it makes it difficult to teach "cognitive" material like planning and knowledge representation, since that material generally presupposes the ability to sense individual objects.

The first three points argue for a "small" robot, i.e. one less than 50Kg. The latter two argue for a “fancy” robot, which often means a big, heavy, dangerous robot. We have tried several different configurations over the years, none of which has been completely satisfying.

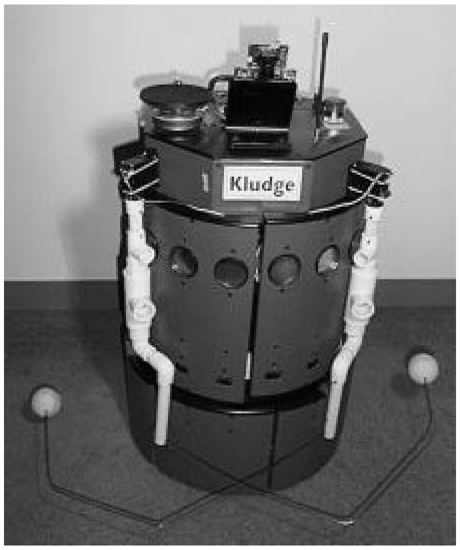


Figure 1: Kludge: An RWI B14 with real-time vision system

The first four years of the course were taught with two Real-World Interface B14 systems, which we had shortened to increase mechanical stability at high speeds (figure 1). The robots were controlled by a custom DSP/framegrabber combination (the DIdeas Cheap Vision Machine), for which we wrote a boot ROM with a variety of active vision subroutines and a simple LISP interpreter for scripting and debugging. LCD displays were mounted on top of the robots to display debugging information.

These systems were probably the easiest of all our robots to program. You turned one on, pushed reset, and it was booted and ready to go within 500ms. If your program crashed, you dropped into a debugger. If the actual hardware crashed, there was a freespace follower in ROM that you could use to herd it back to home base. An important feature of this system was that the Lisp interpreter had a number of hard-coded safeguards to prevent students from damaging the robot. It implemented acceleration and velocity caps that could not be overridden. It also had low-voltage and over-current interrupts that would kill the students' programs if they tried to abuse the batteries. However, a serious disadvantage of the machine was that you could only conveniently program with the on-board Lisp interpreter. Writing new vision operators required off-board compilation and linking of C code, which was then burned into a ROM and tested. This is not for the faint of heart.

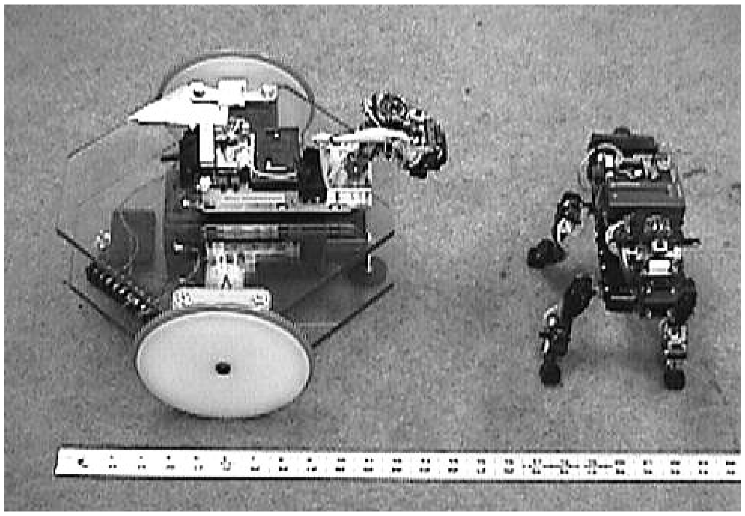


Figure 2: Sony robot with homemade wheeled base

Eventually, the DSP cards in the B14s began to die and we needed another solution. Our initial choice was to develop our own platform based on a DEC StrongARM evaluation board with a Quickcam. We believed that we could build $1,000 platform that would support real-time vision and scripting in Scheme. However, we were unable to obtain the evaluation boards. We also discussed using the Sony Aibo system, and ported most of our code to it (figure 2). However, Aibo development kits proved to be too scarce to support the class, so ended up writing a second port of our code to a cheap single-board computer (based on the Wilke Technologies BASIC Tiger) running on an RWI Magellan base. These had the advantages of being cheap and plentiful, but the serious disadvantages of not having a video display for debugging, and being too slow to support vision.

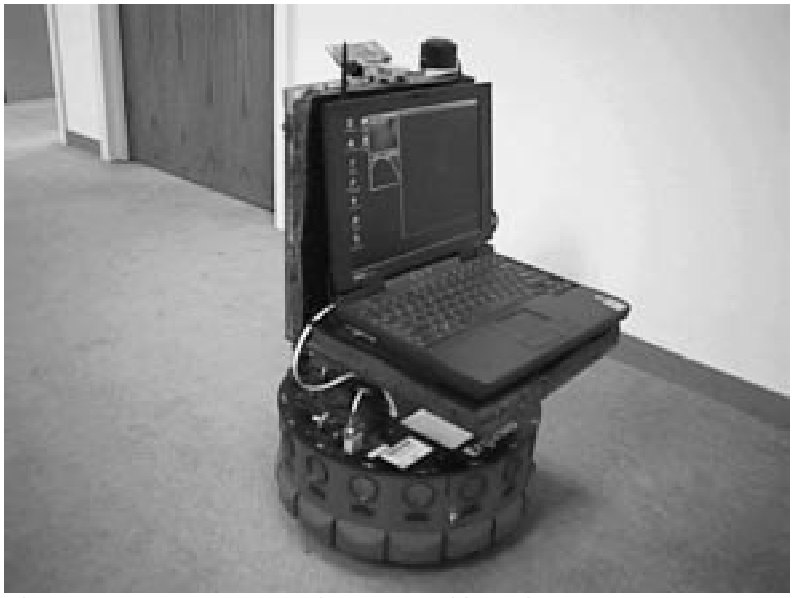


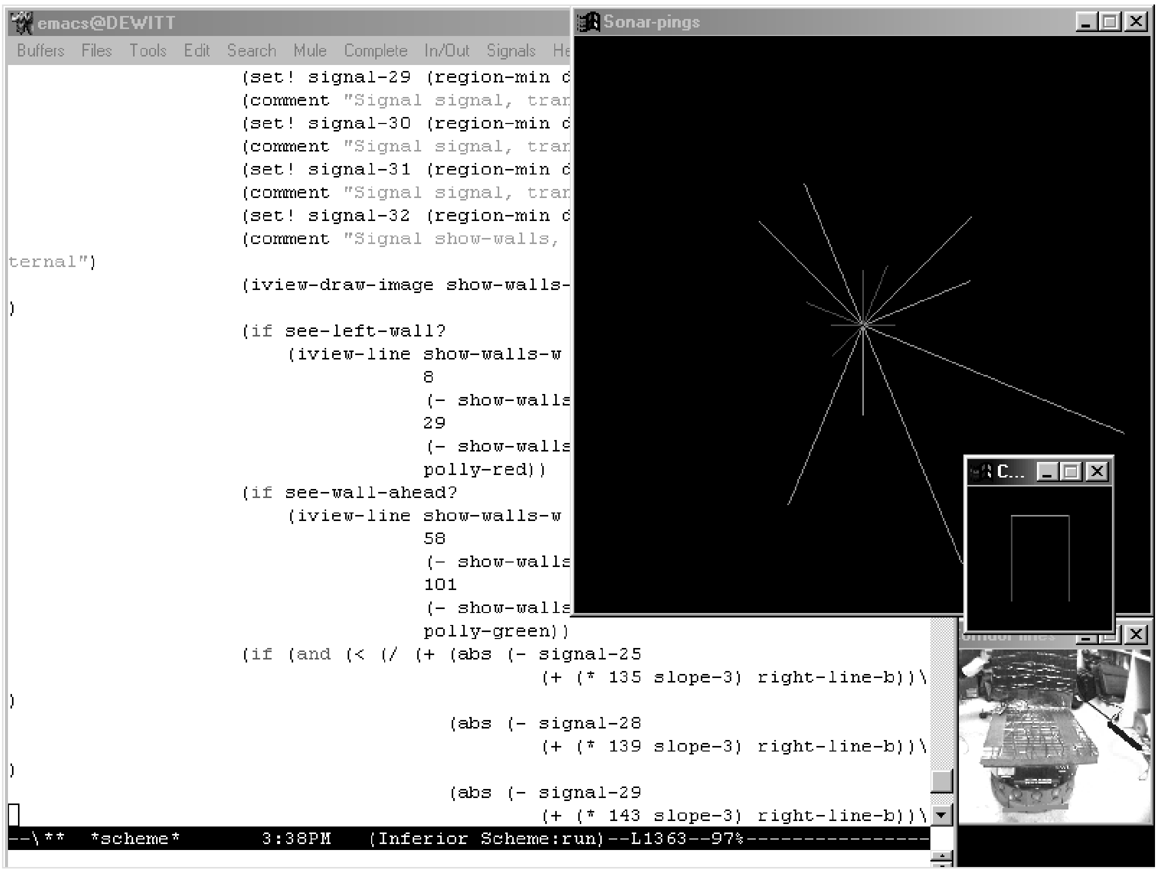
Figure 3: RWI Magellan w/Laptop and board camera

We have now re-re-ported our code to Windows 98. Our current hardware configuration consists of a Magellan base with a commercial laptop (Dell Latitude), a USB frame grabber (Nogatech), and a CCD board camera (figure 3). This is a relatively expensive configuration (about $10K), but most of the cost is actually in the base. A cheaper base could probably be substituted since the sonars on the Magellan are really only used as backup for the vision system.

Programming environment

As we said before, it is critical to provide real-time displays of sensor data and internal state for debugging purposes. Without it, students will typically just try to debug their programs by trial and error. At the end of an assignment, they often can't explain why it is that their programs work (or don't). Our experience with the different hardware platforms underscores this. The B14+DSP configuration and the Magellan+laptop configuration were considered easy to program, while the microcontroller configuration was regularly cursed by the students.

We also felt that it was important to have an interactive programming environment. We specifically wanted to support some dialect of Lisp, since it is the language of choice for AI courses and we wanted to integrate an AI curriculum into the robotics curriculum. The Lisp system on the B14s was a simple interpreter written over a weekend by the author that took up about 8K of the boot ROM. It didn't support macros, or even garbage collection, but it was still a big step up from cross-compiled C code.


emacs@DEWITT
Buffers Files Tools Edit Search Mule Complete In/Out Signals
Sonar-pings
C...

```
                     (set! signal-29 (region-min c
                     (comment "Signal signal, tran
                     (set! signal-30 (region-min c
                     (comment "Signal signal, tran
                     (set! signal-31 (region-min c
                     (comment "Signal signal, tran
                     (set! signal-32 (region-min c
                     (comment "Signal show-walls,
ternal")
                     (iview-draw-image show-walls-
)
                     (if see-left-wall?
                         (iview-line show-walls-w
                                     8
                                     (- show-walls
                                     29
                                     (- show-walls
                                     polly-red))
                     (if see-wall-ahead?
                         (iview-line show-walls-w
                                     58
                                     (- show-walls
                                     101
                                     (- show-walls
                                     polly-green))
                     (if (and (< (/ (+ (abs (- signal-25
                                               (+ (* 135 slope-3) right-line-b))\
)
                                       (abs (- signal-28
                                               (+ (* 139 slope-3) right-line-b))\
)
                                       (abs (- signal-29
                                               (+ (* 143 slope-3) right-line-b))\
--\**  *scheme*       3:38PM   (Inferior Scheme:run)--L1363--97%---------------
```

Figure 4: Screenshot from the current programming environment

Our current programming environment is based on the Scheme48 system of Jonathan Rees and Richard Kelsey [5]. It is a small byte coded system that runs comfortably in 2Mb and can be forced to fit into as little as 100Kb [6]. We've ported the system to Win32 and linked in our vision and graphics code via a foreign function interface, allowing students to experiment interactively with behaviors and graphically display their internal states (figure 4).

Programming language

The course was initially taught in Lisp. However, we found that students had difficulty transferring the concepts discussed in lecture to their laboratory assignments. In the literature, behavior-based systems are most commonly described as parallel networks of simple computational elements (adders, multipliers, finite state machines, etc.) communicating over fixed signal paths. However, what gets drawn as a box an pointer diagram on the chalkboard gets implemented in practice as a while loop containing a series of assignment statements in which each signal from the box-and-pointer diagram is successively updated and stored in a variable. For example, a behavior that drove the translate motor with a low-pass filtered version of some sensor signal might get implemented in C as something like this:

(Code fragment 1)
```
while (1) {
  …
  sensor = … code to read sensor value …;
  …
  filtered_sensor = k1*sensor+k2*filtered_sensor;
  set_translate_velocity(filtered_sensor);
}
```

This is fine, so far as it goes, but suppose we wanted to implement a separate behavior that overrides this behavior and stops the motor when some other sensor returns a reading over threshold. Students will almost always write the code like this:

(Code fragment 2)
```
while (1) {
  …
  if (sensor2() > threshold)
    set_translate_velocity(0);
  else {
    …
    sensor = … code to read sensor value …;
    …
    filtered_sensor = k1*sensor+k2*filtered_sensor;
    set_translate_velocity(filtered_sensor);
  }
}
```

or, more readably:

(Code fragment 3)
```
while (1) {
  …
  if (sensor2() > threshold)
    run_behavior1();
  else
    run_behavior2();
}
```

where `run_behavior1()` and `run_behavior2()` are defined as the bodies of the consequent and alternative of the conditional in fragment 2. Unfortunately, neither of these examples is correct because they only update the internal state of the low pass filter when behavior2 is running. This leads to bizarre transients when the robot switches modes. These kinds of bugs are very subtle and most students can't find them. While there are ways to avoid running into these problems by cleaver structuring of the code, most students do just the opposite – they learn that breaking their code up into separate procedures is bad and so they write huge blocks of spaghetti code which quickly becomes unmaintainable. Moreover, any sense of the program being structured as behaviors and combinators gets lost because there are no objects in the source language that correspond to behaviors or to combinators.

We eventually dealt with this problem by designing a specialized language, called GRL, for writing behavior-based systems in which signals and functions over signals can be composed in a natural functional manner. In GRL, a correct version of fragment 3 would be written simply as:[1]

```
(prioritize (behavior (> sensor2 threshold) 0)
            (behavior #t (low-pass-filter sensor1)))
```

A full description of GRL is outside the scope of this paper. The interested reader should see [4]. What matters for our purposes is that finite-state transducers like low pass filters, as well as behaviors, and the operation of overriding one behavior with another are all first-class data objects in the language and compose in the natural manner.

The disadvantage of introducing GRL was that it meant that the students had to spend time learning the language, and more importantly, learning to think in the language. However, once we introduced GRL into the curriculum, large classes of bugs, like the one in fragment 3, simply stopped appearing. Although it is still difficult to convince students to structure their code as discrete behaviors and arbitration mechanisms, it is considerably easier to do when the students are writing GRL code.

Conclusions

Mobile robotics is a field with inherent appeal for most engineering students. It is highly successful at motivating students to work. One year, 30% of our class was in the lab working on Superbowl Sunday. This was surprising given that the lab is located half a mile off campus and there was a snowstorm going on. It was mildly stupefying because we hadn't given them any assignments to work on. They wanted to get their previous assignments to work better, even though it wouldn't affect their grades.

Sadly, students are not always motivated to work on the things that we as teachers feel they should want to work on. Again, we found that students tended to want to make their robots run as fast as possible, even when they knew it was counter-productive. The only way we found of overcoming this tendency was to give them a different criterion to optimize. We also found that students weren't necessarily interested in trying out motor schemas, layered architectures or any of the other interesting theoretical constructs we discussed in class. They just wanted to win the contest. Even more irritating was the fact that the winners usually weren't the ones using the nice theoretical constructs. This situation was helped by giving them a programming language that makes it easier to write these constructs, but it can still be an uphill battle to get them to write structured code.

Finally, it should be noted that although demos and contests are valuable motivators for many students, they are intimidating for others. Contests are good for the winners, but embarrassing for the losers, particularly if they already have low confidence as programmers. In general, we think it is useful to structure assignments more as "talent shows" where there are a few different ways in which students can be perceived as doing well (e.g. high speed, pretty code, and general coolness) and so there are no absolute winners or losers.

Acknowledgements

This work was made possible in part by a grant from the National Science Foundation CAREER program. I would like to thank the Ben Slivka, Lisa Wissner-Slivka, the Microsoft Corporation, and the Sony D21 laboratory for their generous support. I would also like to thank the people at the Real World Interface division of IS/Robotics for their high level of tolerance for fixing our robots no matter what the undergrads did to them. Most of all, I'd like to thank the students of the Autonomous Mobile Robotics Group at Northwestern who worked to make the course possible: Chris Beckmann, Maj. Pete Beim, Lars Bergstrom, Matt Brandyberry, Mark DePristo, Aaron Khoo, Dac Le, Karl Marsilje, Maj. Clifton Poole, Ivan Yen, and Robert Zubek.

[1] It should be noted that `prioritize` is really called `behavior-or` because it behaves like the `or` operation in Lisp.

References

1. R. Arkin, Behavior-Based Robotics, MIT Press, May 1998.
2. A. Druin and J. Hendler, Robots for Kids: Exploring New Technologies for Learning, Morgan-Kaufman, April 2000.
3. I. Horswill, “Polly: A Vision-Based Artificial Agent,” *Proceedings of the Eleventh National Conference on Artificial Intelligence* (AAAI-93). Washington DC, 1993.
4. I. Horswill, “Functional Programming of Behavior-Based Systems,” *Autonomous Robots,* Kluwer Academic Publishers, Norwell, MA. In press.
5. R. Kelsey and J. Rees. “A Tractable Scheme Implementation,” in *Lisp and Symbolic Computation* 7(4), 1995
6. J. Rees and B. Donald. “Program Mobile Robots in Scheme.” *Proceedings of the IEEE International Conference on Robotics and Automation,* pp. 2681-2688. Nice, France (May 1992).