Chapter 4

Ideal and Real Systems

The product of each robot design project is a particular robot that interacts with its environment—the contest playing field, the game objects, and the opponent robot or robots—in particular ways. The robot thus embodies and becomes part of a complex system of interactions that defines its existence.

In this chapter I will explore the biases that students bring to the task of designing their robot, a member of such a system. Of particular interest is students' recurring inclination to build robots that will perform properly only under ideal conditions. Students repeatedly build robots that are not well-equipped to deal with the exigencies of the real world, but rather with the specifications of an idealized, abstractified world—a world that the robot designers would like to believe is a close representation of reality, but is not. This result points to limitations in the set of ideas about technological systems and methods that comprise the core of the engineering curriculum. What surprises many participants is that these ideas do not map well to the challenge of designing a robot to play in one of our contests.

This chapter has four sections. The first section presents an overview of the task of designing a control system for a contest robot, and the sort of ideas that students generate in coming up with their own solutions. This section presents a composite set of results from a number of different students' work; this composite portrait is representative of the sorts of issues encountered by most of the students in the classes.

The second section presents a case study of a pair of students who formed a team to build robots for two consecutive years of the Robot Design class. These students believed that by control could best be achieved by having the robot keep track of its global position on the playing field at all times. In the face of failures, these two students' views held with a greater degree of clarity and perseverance than most, yet in important ways they are "cut from the same cloth" as others.

The third section presents the approach taken by one student who developed the premise of simulation as a robot development tool. While this student's methodology was somewhat unusual, his approach reflects biases showed by many of the students.

The methodologies chosen by the student robot designers in this study are in many ways a product of the curriculum of the modern engineering university. The final section analyzes the content of this curriculum, pointing to evidence that explains students' choices, and why their intuitions about what sort of control would be effective were largely wrong. In doing so, I point toward a direction for revising the curriculum to encompass a broader range of ideas about systems and control—one that would be more effective in preparing students for demands of modern technological systems.

4.1 Introduction to Robotic Control

The data for the issues discussed in this section is taken primarily from the students' work in the *Robo-Pong* and *Robo-Cup* design contests. In these two versions of the project, both the design task and the materials used by the students were sufficiently rich to allow a variety of approaches to become manifest. In contrast, the *King of the Mountain* and *Robo-Puck* contests were too simple for these issues to arise.

4.1.1 The Omniscient Robot Fallacy

Students' beliefs in how a robotic system should be understood and controlled is revealed in how they approach the central task of developing their robot's strategy.

Some students approach the design of the robot task from what might be called an egocentric perspective, as if they were thinking, "what would I do if I were the robot."

These students imagine themselves at the helm of their robot, driving it around the table and performing tasks. This approach leads to ideas like the ones in the following narrative:[1]

> Our strategy at present seems fairly well developed. First, [our robot] will immediately locate the other robot, roll over to it while lowering the forklift, and try to flip the other robot over. If it can knock it over, or even over the playing field wall, the robot has essentially won, as it can now freely place balls in the goal without interference. If this attempt fails after ten seconds or so, our robot will disengage, roll over to the ball dispenser, park, and get balls and shoot them at our goal with a cannon mechanism. Should the cannon prove unreliable, we may have to collect balls and deliver them manually, but having a cannon would be much more nifty, and would avoid a lot of line- or wall-following difficulty.

It is apparent in this discussion that the student has not thought through, at a mechanistic level, the questions of how the robot might actually perform these tasks; he or she is imagining that it is simply a question of navigating as one might drive a car.

Many students initially generated such fantastic ideas about what sorts of activities of which their robot might be capable. Here is another student's ideas about how to solve a contest:

> I played around with a structure that was to be the "Goal Emulation Unit," which was an idea of mine. We had decided it would be a neat strategy to have something on our robot that looked like a goal, so that it could stand in front of the opponent's goal, and hopefully the opponent would place its balls into our robot's unit . . . We had decided in the beginning that we wanted a fast robot. I thought it might be neat to chase after our opponent as soon as the round started and either steal his ball (assuming that it goes to the dispenser first) by pushing him out of the way, or having some sort of long arm that reached above the other robot to get the ball.

Written at the end of the second week of the project, these two scenarios are representative of students who have difficulty imagining what sort of information with which the robot will be working, and hence what sort of strategies or algorithms will yield realistic performance results.

Quite a large number of students initially generate ideas like these. If an idea requires a complex mechanical structure, such as the "Goal Emulation Unit" described in the previous

[1]The narratives in this and the following chapter are taken from the students' written design reports.

passage, students will discard their ideas as soon as they're unable to build the mechanisms that are required to accomplish it (which is typically early in the project). If the complexity resides in the software design, however, many students continue to imagine their robot to be capable, in principle, of very sophisticated behaviors—until they attempt to implement those behaviors in actual code. For example, in the following comment, a student believes his robot to be all but completed, even though he has not began programming it:

> As my team's robot stands now, it is 99% near completion. Only a few structural changes (might) have to be made and one or two sensors have to be attached; otherwise all that is left to do is program the baby.

Evidently, this student is not including the programming task in his estimate of the robot's percentage of completion. This attitude is typical; not only do students consistently underestimate the time-consuming nature of the programming task, but until they attempt the programming task, they don't realize what sort of robot behaviors are feasible to implement.

Other students more readily acknowledge the unknown nature of the programming task. Most fall back on previously learned approaches for dealing with complexity, as this student explains:

> In addition to assembling part of the robot base, I have been working with Interactive C, writing some code but mostly pseudocode, and relying heavily on abstraction and wishful thinking.

This student has not gotten into the hard work of getting his robot to perform real behaviors with but one week before the contest.

A second group of students is more cognizant of the practicalities of the issue, realizing from the start that they need to get a firm grasp on their robot's potential capabilities. After describing a number of different strategic possibilities, one student writes this statement after the first week of work:

> However, [my teammates] and I realize that the only way to find out what strategy works best is to build hardware that is robust and is powerful enough to implement a lot of different strategies and then implement the actual strategies in software. Our immediate goal is to construct powerful hardware both electrical and mechanical and then write the software to control it and implement the different strategies.

This hands-on approach to developing a robot's program is favored by fewer of the students.

4.1.2 Understanding Sensing

Few participants in the class have had prior experience working with electronic sensors. When they are first introduced to sensors, either through the lecture-style presentation we hold, or in the course notes, many students expect that sensors will provide clear and unambiguous information about the world. As one student explains:

> We plan to build the sensors and motors first, so that I can start playing with the control software and writing subroutines so that we can abstract away the working of the sensors as much as possible.

This student speaks for many when he proposes the use of modular abstraction to deliver clean information about the robot's state to some higher level control program. However, there are many reasons why sensors don't work as straightforwardly as the students would like; some of these have been discussed in Section 3.3. Students are then surprised when they attempt to program a functional behavior into their robots. As one explains:

> I decided to write the wall-following function which was the example in the text. It turned out to be much more difficult than the text had led me to believe. First of all, I needed to figure out how much power went to the inside motor in a turn. This took more than a few hours to finally debug and figure out what kind of radius each power differential would give. Another thing I had to figure out was the thresholding and logic statements needed for two sensors, one at the front and one at the back. This thresholding problem took me more than a while to figure out, and in the process, I found errors in my logic statements. . . This method has not proved anything, because in its first incarnation, the robot was low on batteries. . . [the] present program hasn't been tested yet, because. . . we took the robot apart to strengthen each subsection, as in the wheel assemblies, and the subframe to connect them.

In the end, this student has concluded that his tests are moot, since the robot had a low battery level when the tests were being done. Even if he hasn't solved the particular problem at hand, though, it is clear from his narrative that he learned a great deal about the nature of the problem situation.

Here is another account of sensor troubles written by a participant who began systematic sensor experiments before the "crunch time" of the end of the project (and hence had time to write about it):

> The sensors have plagued me with problems from the start. The photoresistor sensors work great and I think that we can rely on them for the polarized light sensors. However, I think that the polarized light filters are not very reliable in tests that we have been using. It seems that the photoresistor in the polarized light sensor can't tell the difference between facing the opposite goal and being far away from its own goal. I think the problem lies in ambient light reaching the photoresistor. I think the only way to solve this problem is better shielding, but I think that the ambient light problem is unsolvable unless the contest is [held] in a pitch black room.
>
> The reflector sensors seem to work all right, but when we try to shield them from ambient light, we get random reflectance and therefore detection when we aren't supposed to. The problem seems to be solved when we place the sensor inside my sweater which is very "bumpy" and keeps light from randomly reflecting off of things. We are going to try to fix this problem this week because our strategy really depends on them.

These two cases are unusual only in that the respective work took place early in the course of the design project. They are quite typical, however, of what nearly all of the students encounter when they engage in the actual programming task. Students then discover all sorts of interrelations and complexities that arise, aside from implementation issues, that make it difficult to abstract sensor data into clean information upon which control strategies can be devised. They realize that sensor data are erratic; that there are dependencies on hard-to-quantify properties of the mechanical system; and that how one solves the control problem is a factor in the ultimate reliability of the sensor data. For example, here a student finds the solution to achieving a good edge-following behavior lies in accomplishing several changes:

> When we first tested the robot's ability to detect a green/white boundary, the robot had difficulties. After some software changes (by [my partner]) and mechanical changes by me of placing the sensors even closer to the ground and also placing shields around the sensors, we have no problems detecting the boundary.

Students had a tendency to establish hard-coded thresholds for interpreting the meaning of sensor data. In *Robo-Pong*, two sensor varieties were particularly problematic in this

regard: motor current sensing, which determined if a robot's movement was impeded (causing its motors to stall and hence draw more current) and light sensing (for determining on which side of the playing table the robot was located).

The trouble with the motor current sensing was that the numerical reading that indicated a stalled motor would change as a function of the voltage of the motor's battery. When the battery voltage fell, as the battery gradually discharged with use, the current sensor would return higher values—indicating that a motor was stalled when it in fact was not.

Several teams of students failed to adequately deal with this problem and created robots that acted as if they were stuck when, in fact, they were just driving uphill. This occurred because uphill driving would necessarily cause additional load on the motors, which the robots' programs would interpret as the case where the robot was indeed stuck. In the main contest performance, one robot in particular suffered an amusing yet incapacitating failure mode of this type. It would repeatedly drive forward, stop, and back up as its program interpreted the motor current sensors as indicating that the robot was stuck. The robot seemed to be battling an invisible enemy as it tried again and again to collect the balls in its trough, each time backing up after advancing a few steps.

Students in *Robo-Cup* were warned of this problem and some concluded that the motor current sensing was simply too unreliable to be a part of their design. As one student explained:

> I wrote servo-like routines to monitor the clamp's position, and attach or release it as required. The original plan was to use the motor force sensing to determine if the motor was stalled, and control its motion like that. The force sensing, it turns out, it very unreliable, so we chose to install a potentiometer [a rotational position sensor] on the axle instead, and use that to keep track of position.

This student has performed an analysis that this sensor is too erratic to be trusted, but most did not discover the sensor's failure mode. Students simply did not anticipate the sensor's fundamental unreliability.

The light sensors used in *Robo-Pong* were problematic in that they relied on room lighting to illuminate the table playing surface. Hence the values registered by sensors—of the amount light reflecting off of the playing surface—were dependent upon the amount of ambient light in the room.

We as organizers were aware of this situation when we designed the contest and chose sensors for the robot-building kits, but we underestimated the impact on students' robots that resulted from changing the light levels for the contest performance. Anticipating that the lighting conditions would be different on the eve of the contest than they were in the development laboratory (since we intended to use camera lights), we had informed students that they should either shield their sensors from ambient light and provide a local source of illumination (e.g., a light emitting diode or flashlight bulb), or write calibration software that would take into account the level of ambient lighting.

However, we failed to provide a readily accessible way for students to try different lighting conditions during robot development, and the camera lights we used were particularly harsh, causing wide fluctuations in light levels even from one side of the table to the other. It was an unfortunate event as a number of otherwise competent robots became unreliable in the face of the extreme lighting conditions.

This situation led us to propose to a number of changes in *Robo-Cup*. First, we provided a light sensor that incorporated its own light transmitter, so that it would be substantially easier for students to deploy light sensors that were well-shielded from ambient light. Second, we incorporated a diffuse lighting fixture into the contest playing field itself, so that lighting conditions would not change drastically from development to performance situations. Also, we modified the programming environment to make it easier for students to construct sensor calibration routines: we made it possible for calibration settings to be "remembered" throughout multiple performance runs.

Nevertheless, students continued to have similar difficulties dealing with the sensor calibration issue. Its implications were underestimated and the concept seemed unfamiliar to them.

4.1.3 Models of Control

The overall strategies that students used to control their robots lie in a spectrum between two categories, which I shall call *reactive* strategies and *algorithmic* strategies. In addition, each of these strategies is built from two lower-level methods, *negative feedback* and *open-loop* approaches. Analysis of the various approaches that students take toward implementing

an overall strategy to control their robots reveals interesting biases, as was the case in analyzing students' approaches to understanding and deploying sensors.

These descriptions that categorize the students' robots were created in an analysis after the fact of the students' work on them; students were not aware of these categories during the course of their projects. I believe that students would recognize these distinctions if asked, however.

The Higher Level: Reactive and Algorithmic Control

The issue of control manifests itself in at least two aspects of the Robot Design project: in the development of the robot's higher-level strategy from a conceptual standpoint, and in the actual programming of that strategy into the machine through the process of writing computer code. In the conceptual area, the *algorithmic* control method is by far dominant over the *reactive*.

The algorithmic method is characterized by a program that dictates a series of actions to be taken as its central feature. For example, here is an algorithmic main program loop from a student's solution to *Robo-Pong*. The code waits for the starting lamp to turn on (`start_machine`), then executes a routine to turn the robot downward (`rotate_down`), executes a routine to sweep across the ball trough (`grab_balls`), waits for two seconds, and then drives to the other robot's side of the table (`other_side`):

```
main()
{
    start_machine();  /*starts the machine up when*/
                      /*the light is on*/
    rotate_down();
    grab_balls();     /*go into our grab balls routine*/
    sleep(2.0);       /*wait two seconds*/
    other_side();
    finish();         /*block other machine or */
                      /*knock more balls over?*/
}
```

In this program, the contest is solved by executing a specific series of actions. Here is another example of the same approach, albeit slightly more complex, for solving *Robo-Pong*. The names that the student has given to the subroutines of his program are fairly self-explanatory as what action each subroutine performs:

```
game()
{
    start();
    Turn_to_Uphill();
    Go_Up_and_Orient_Plateau();
    DoorOpen();
    WalkPlateau();
    DoorClose();
    JiggleLeft();
    DoorOpen();
    FollowRightWall();
    DoorClose();
    JiggleLeft();
    DoorOpen();
    FollowRightWall();
    DoorClose();
    JiggleLeft();
    FollowRightWall();
    AllOff();
}
```

Code written using the same algorithmic method can be found for *Robo-Cup* robots. Here is an example of such a main routine:

```
starting_routine()
    {

    /* drive forward full speed */
    drive(7);

    /* wait until sense edge */
    while(abs(ana(reflectance0)-white0)<40) {}

    /* drive a little further */
    sleep(distance(2.0));
    alloff();

    /* turn a little */
    pivot_on(7);
    msleep(80L);
    pivot_off();

    /* drive until front bumper triggers */
    while(!(digital(front_bumper)))
    drive(7);msleep(20L);
    drive(0);

    collect_balls(3);
    goto_goal();
    goto_button();

    collect_balls(3);
    goto_goal();
    goto_button();

    catch_ball();
    goto_goal();
```

```
        goto_button();
        return;
    }
```

As illustrated, the algorithmic control methodology consists of a list of instructions to be followed in order to accomplish the intended task. Each instruction may consist of a set of sub-actions, but the overall principle is to decompose the task into a sequence of activities to be followed, much like one might construct an algorithm to sort items in an array or compute a statistical quantity.

The trouble with the algorithmic approach is that there is no recovery path if things do not proceed according to the plan. Not only is there no way to recover from unexpected circumstances, but there simply is no provision for detecting that something out of the ordinary has occurred. These solutions have a problem that is reminiscent of comical factory scenes in which the machinery continues to plug away at an assembly process that has gone completely awry.

Students often deceive themselves as to the reliability of their algorithmic solutions. Suppose each component of an algorithmic solution has a 90% likelihood of working properly on any given occasion. One might think that the overall solution would have a similarly high probability, but this is not the case, since the probabilities multiply together with the overall result decreasing in likelihood with each step. If the algorithm has six independent steps, then the overall probability of proper functioning is only 53%—not nearly as good as the 90% chance that each individual step has of working properly. During testing, an algorithmic solution might require a gentle prod or correction here and there, and students don't realize that their machine actually needs this sort of help a significant portion of the time.

The reactive methodology, on the other hand, is characterized by actions that are triggered through constant re-evaluations of external conditions (i.e., sensor readings) or internal conditions (program execution status). While the algorithmic method does make use of external stimuli, the reactive method is more *driven* by this data, rather than by using it as part of a predetermined activity sequence.

There were no purely reactive robots in any of the contests; instead, some students combined elements of a reactive strategy into an overall algorithmic framework. An

example is the robot *Crazy Train*, which was the champion of the *Robo-Pong* contest. The robot was a collector-style machine that scooped balls into its body.

Crazy Train worked as follows. As the round began, it took a reading from its inclination sensors to determine its initial bearing. It if determined that it was pointed toward the top of the hill, it would drive forward until it crossed the center plateau; in the process, it was likely to knock a center ball onto the opponent's territory (thus gaining an early advantage—insurance in case something went wrong later in the round). If it was not aimed uphill, or, after having executed the initial upward movement, it would drive to its own ball trough and collect balls into its body. After making one sweep of the trough, it would drive onto its opponent's side of the table, delivering the balls it was carrying onto the opponent's side. In addition, when ten seconds remained in the round, *Crazy Train* checked to see which side of the table it was presently on. If for some reason it was still on its own side of the table, it would execute a "panic" routine which would attempt to drive onto the opponent's side before the round ended.

There are two aspects of *Crazy Train*'s program that characterize the reactive method. Firstly, *Crazy Train* had an unusual response to sensing its orientation at the start of each round. Depending on the orientation, the robot choose whether to go for the potentially risky maneuver of immediately knocking a ball over the top, or the safer maneuver of traveling to its trough to collect balls. It chose to go for the early lead by knocking over a center ball only if it is was efficient maneuver—when the robot was already aimed toward the center plateau. In this fashion *Crazy Train* reacted to an external condition and choose an appropriate response.

The other feature of *Crazy Train*'s program is its panic behavior when the contest round was about to end. Here the main program, albeit an algorithmic one, is interrupted by another behavior which is triggered by a change in internal condition—a timer keeping track of remaining contest time. If necessary, the panic behavior attempts to drive onto the opponent's side (if the sensor registers that the robot is already on the opponent's side, which should be the case if the algorithmic task has been completed successfully, then no additional action is needed). The students who created *Crazy Train* implemented the panic behavior by exploiting Interactive C's multitasking capability. Their program had

both the algorithmic behavior and the panic behavior as independent program tasks; at the appropriate time, the algorithmic task was terminated and the panic task was initiated (a third program task was used to coordinate this activity).

Crazy Train is perhaps the most dramatic example showing the usefulness of reactive program strategy, and, not too surprisingly, it was an overall contest winner. Its success can be attributed to a combination of its reliable and effective ball-gathering algorithmic strategy bolstered by useful reactive behaviors: there were a few game rounds in which the algorithmic strategy failed but the reactive behaviors delivered a winning performance. It also is representative of the extent to which reactive behaviors were used at all; that is, where reactive methods were used, they were simplistic, though sometimes effective, responses to a limited number of potential situations.

The Lower Level: Negative Feedback and Open-Loop Methods

The issue of reactive versus algorithmic control as it applies to the macro level of overall robot strategy has its analog at the micro level, the mechanism by which the strategy is actually carried out. The micro level consists of the low-level methods for driving the robot's motors to accomplish an intended action.

For example, the macro tasks of collecting and delivering balls in both *Robo-Pong* and *Robo-Cup* can be decomposed into micro- or sub-tasks such as climbing uphill or downhill, driving along the retaining wall of the playing field, or driving along a light-dark boundary painted on the playing field floor. The contests were specifically designed to be decomposable into subtasks like these—ones that are feasible to be solved with the technology provided in the robot kit.

I distinguish two low-level control methods that were implemented in the robots. Most robots used a mixture of these two methods rather than one or the other exclusively.

The two methods are *negative feedback* and *open-loop control*. In the negative feedback method, a robot responds to a situation with a small corrective action. Through repetitive application of the small maneuver, the robot achieves the overall action desired by its designer. For example, a robot might follow along the edge of a wall with the use of a sensor that measures the distance to the wall, repeatedly turning to move toward the wall if

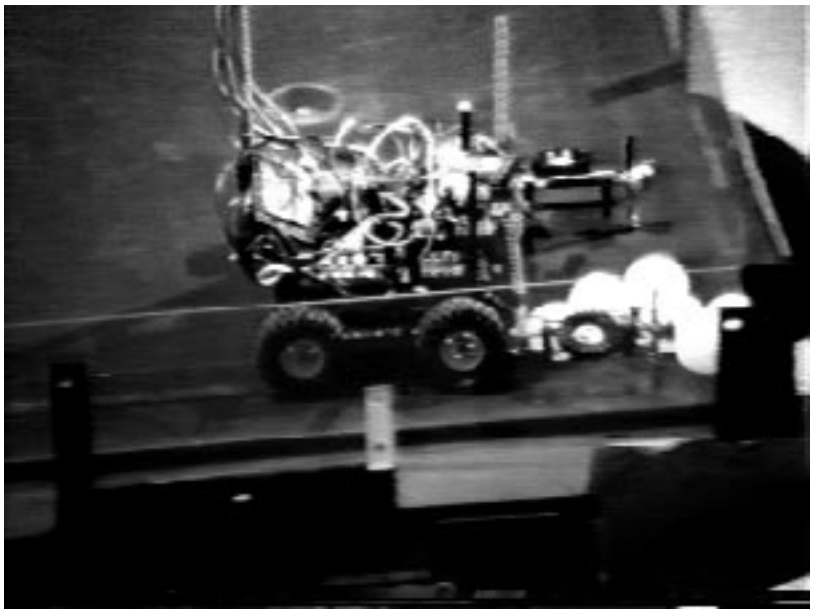


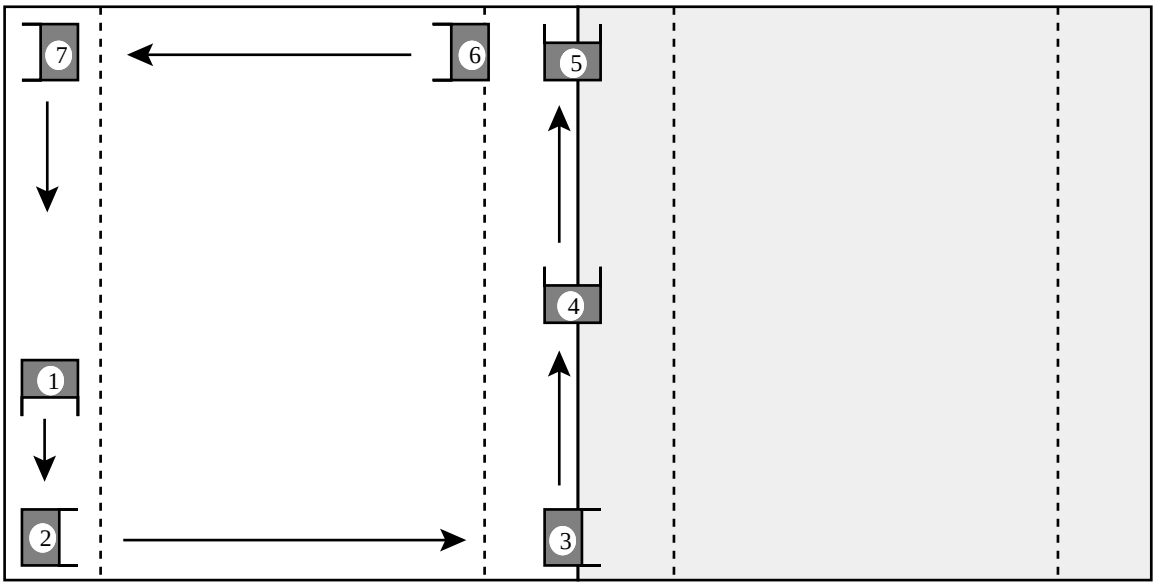
Figure 4-1: Photograph of *Groucho*

it senses that it's too far away, or turning to move away from the wall if it's too close.

In the open-loop method, the robot responds to a situation with a single action which the designers presume will bring it to a new state with respect to its surroundings. A typical action would be a predetermined, timed motor movement, usually lasting one or more seconds. For example, the robot might turn its motors on for a period of two seconds (this value would be experimentally determined) in order to pivot ninety degrees and thereby negotiate out of a corner.

Few of the robots' activities can be considered open loop in the formal sense because the open loop-like maneuvers are part of a larger robot strategy that generally involves sensor input and hence is not truly open loop. However, if a robot takes an action that does not accomplish the desired result, and the control algorithm cannot recover gracefully from this circumstance (nor makes an effort to detect it), the action is likely to be representative of open-loop thinking.

An example will help make these ideas more clear. *Groucho*, a *Robo-Pong* robot (pictured in Figure 4-1), employed the following strategy. It would drive into its trough at the beginning of the round and then execute a series of motions that would make it drive in a rectangular pattern, scooping balls from its trough and depositing them onto the



1. Robot drives along trough, scooping balls into its grasp.
2. Robot negotiates corner and turns uphill.
3. Robot stops at dividing edge, allowing balls to fall onto opponent's side.
4. Robot drives along edge to opposite side of the table.
5. Robot hits opposite side; turns to drive downhill.
6. Robot drives downhill.
7. Robot hits bottom wall and negotiates corner; continues pattern in step 1.

Figure 4-2: *Groucho*'s strategy as played in *Robo-Pong* contest

opponent's territory. *Groucho* used an overall algorithmic strategy, repetitively performing the sequence of these steps, as illustrated in Figure 4-2.

In implementing this strategy, however, the students who built *Groucho* used mixed techniques of feedback loops and open-loop control. At points 2 and 7, when negotiating corners, they used feedback from the playing field walls to do so: it would hit the wall, back up, make a small rotation, drive forward, hit the wall, and try again. Typically the robot would strike the wall three to five times until it rotated sufficiently.

At points 3 and 5, however, the robot executed timed movements to accomplish the rotations. The designers had experimentally determined how much of a rotational movement, measured in fractions of a second, would be required to perform the desired turn, and then hard-coded these timing constants into their program structure.

The question arises as to why the designers of this particular robot chose to implement a feedback-controlled turn during some points of their robot's path and open-loop-controlled turns at others. There is a clue to the explanation contained in the source code written for the robot. As mentioned, in the trough area, the students used feedback from touch contact to negotiate turns, but in the plateau area, they used timed turns rather than feedback from the center dividing line. It turns out, however, that the students attempted to base the plateau turns on feedback from the dividing line, but at some point commented out the code to do this, replacing it with a simple timed turn. Here is the relevant code excerpt from their program. The line bracketed with "`/*`" and "`*/`" is the one that the students commented out; the subsequent line, `left(100,1.);`, implements the timed turn that replaced it.

```
    printf("Crossed the border.\n");
    beep();beep();
    time=seconds()+1.1;
/*  while ((left_reflectance!=side)&&(seconds()<time))
      {bk(0);bk(1);fd(2);fd(3);} */
    left(100,1.);
    ao();
    printf("On the Line...\n");
```

It can be presumed that some kind of difficulty or unreliability was encountered when using the reflectance sensor to determine when the robot had turned enough, so that the students substituted the open-loop turn movement instead. This robot's mechanics were sufficiently well-designed and reliable that the robot's performance through the turn remained

consistent through the contest performance, but this is not typically the case.

The example of *Groucho* brings out a tension between the negative feedback technique, which ultimately can be more reliable, and the open loop technique, which is often quicker to implement and can result in faster performances, but is more subject to failures. In these excerpts, two students comment explicitly on this matter:

> We decided to simplify our strategy, making our robot more reliable and easier to build. We now plan to have our robot aimed at the dispenser at the outset of the contest. The robot will then move forward until it senses that it has crossed the white/green boundary, using an infrared reflectance sensor. The robot will then turn and press the button repeatedly, catching and shooting each ball into the net. *This strategy involves much open loop control, but greatly simplifies our control system, making the robot more reliable.* (Emphasis added.)

> I discussed with my teammates that if we're going to be a fast machine, then we'll have to implement a lot of open-loop [controls] interleaved with closed-loop feedback. Otherwise if we used mostly closed-loop [methods] to "feel" our way around the playing field, we'd be as slow as our opponents.

These students were conscious of the trade-offs involved, but others were not. Many students used open-loop movements based on timed actions; robots built from these constructs faced problems from a number of sources. The most significant variable was the level of charge in the battery powering the motors. As the movements' durations were experimentally determined, they were highly dependent upon battery performance.

Robots were powered with lead acid rechargeable batteries, which have a characteristic discharge curve as shown in Figure 4-3. As the battery is used, its voltage (which determines the amount of power delivered to the motors) gradually decreases. Often this deterioration was not immediately noticeable, so students were not necessarily aware of the situation as they were developing their programs. Finally the battery would reach a point, the end of its usable life, when the voltage would drop off rapidly and the battery would be obviously "dead." However, during the usable period there would be a significant difference between the initial voltage and final voltage levels. (Some other types of batteries, such as nickel cadmium, have discharge curves in which the voltage level remains nearly constant for the bulk of the batteries' usage, until the level drops off sharply, rendering the battery suddenly useless. Use of these batteries would have made the discharge effect less pronounced.)

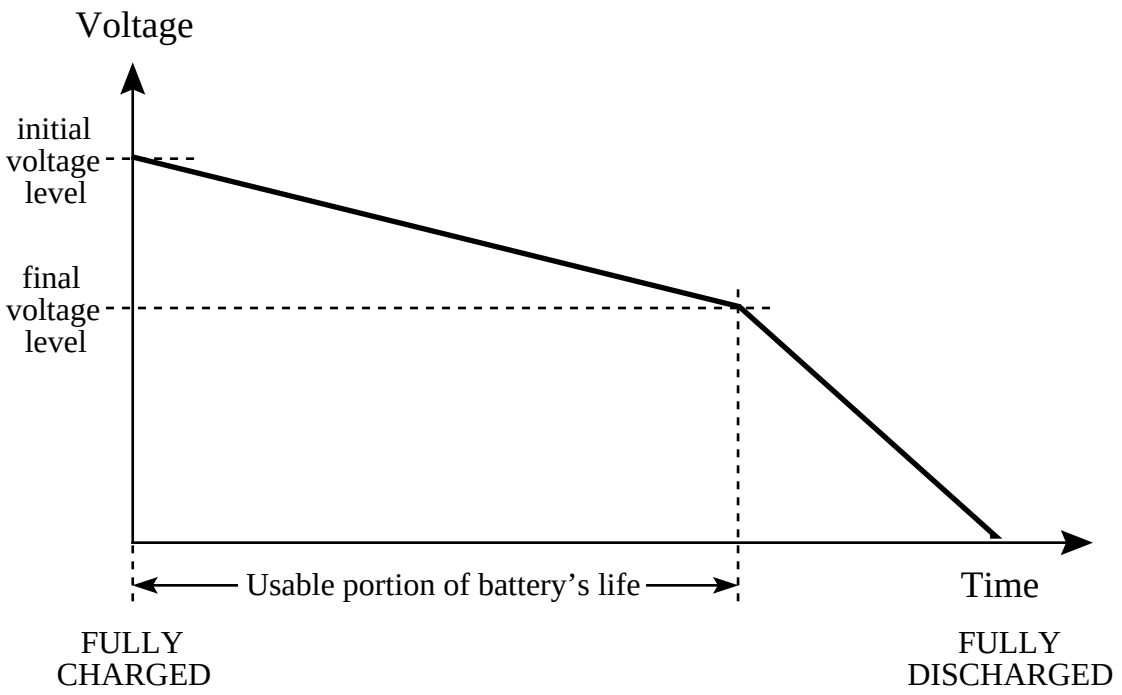


Figure 4-3: Idealized lead-acid cell discharge curve

Most students only realized that the algorithms they had created to solve the contest were dependent upon battery level when they would swap a partially discharged battery for a fully charged one, and discover that their program no longer worked correctly. As mentioned earlier, this realization did not occur until quite late in the robot's development—when there were only a few days or even hours left before the contest—and hence it was too late for them to make a substantive change in the robot's strategy (that is, substituting feedback-based movements for open loop ones). Students would deal with the situation by doing final program development—i.e., tweaking of timing constants—with batteries as fully charged as possible.

The degree to which this problem would manifest itself in the students' designs was largely a function of the gear ratio (that is, ratio between motor rotational speed and final drive wheel rotational speed) used in the robots' mechanics. If the gear ratio was such that the motor was driven in a "comfortable" portion of its power range, a slight change in battery level would not have a drastic effect on the performance of the motor. On the other hand, if the gear ratio caused a high amount of drag on the motor, slight changes in battery level would have a large effect on motor performance.

We as organizers only recognized this situation in retrospect, and hence were as confused as the students as to why some robots seemed to have little difficulty in performing reliably (at least in this regard) while others were quite troublesome. We were more savvy of these control issues when running *Robo-Cup* than *Robo-Pong*. In an attempt to encourage students to use feedback-based control rather than open-loop control, I wrote an additional chapter of the course notes for the *Robo-Cup* students that explained the differences and trade-offs between open-loop and feedback-based control. I also discussed the need for calibration of sensor values to local conditions.

A number of students became fixated on using feedback control in surprising ways, however, spending a considerable portion of development time creating a feedback controller to make their robot able to drive in a straight line. Here is how one student explains his system:

> The robot is now able to drive in a straight line thanks to its optical shaft encoders. The software continually samples both left and right sensors and adjusts the exact power level to the motors to compensate for any fluctuations in the motion of the robot. The algorithm itself is a combination of differential analysis and Newton's method, along with an "adjustable window" of correction. Simply stated, analyzing differentials gives a reasonable algorithm for adjusting the power levels of the motors, and the adjustable window minimizes excessive wobbling. This algorithm gives us as much accuracy as the resolution of the shaft encoders permits, while keeping wobbling reasonably low.

The ability to drive perfectly straight, however, often does little to help a robot in its overall ability to solve the contest task. While the activity of driving straight is based on negative feedback control, a strategy that employed this ability in a central way should be considered open-loop at the next higher level, as the robot would be performing a task with little feedback from the crucial environmental features! So these students' belief that they were using the preferable feedback control was misguided.

Taken as a whole, the collection of biases revealed in students' thinking suggests important misconceptions and misunderstandings about what are effective ways to build reliable real-world systems. The next section builds on this hypothesis, presenting a case study of a student team that tried unsuccessfully to exploit feedback by embedding it in an

open-loop strategy, in a fashion suggested by the students fixated on their robots' ability to drive in a straight line.

4.2 Case Study One: Monitoring Robot Position

In the prior section, I examined ways that students negotiated issues of control in a broad sense, including the higher levels of reactive and algorithmic control and the lower levels of negative feedback and open-loop control. In this section, I present an in-depth case study of one particular team of students. These students brought strong personal convictions about what type of control system would be best for operating their robot. Even after their approach proved unsuccessful, they remained quite attached to it—an indication of the strength of their beliefs.

The students, who I will call "Stan" and "Dave," participated in robot-building teams for two consecutive years of the contest. While all members of the team collaborated on the implementation of each robot, Stan and Dave's strategic ideas dominated the designs. (Their team had other members as well during each of the two years.)

Stan was a master LEGO Technic builder, having been a avid fan and collector of the materials since his childhood and through his teenage years. His ability to design with the materials was quite impressive in both artistic and functional senses. He also was driven to create spectacular and elaborate designs that would have great audience appeal.

Dave, on the other hand, was an accomplished computer programmer, who reveled in building sophisticated and elegant computer programs. Both Stan and Dave were pleased to defer to each other's strengths during implementation of their ideas, but collaborated during the conceptual design stages.

In *Robo-Pong*, Stan and Dave's team chose to build a shooter robot. The task of building an effective shooting mechanism was a genuine challenge even for the experienced LEGO designer; there were a number of teams who gave up on building shooting robots after being unsuccessful in developing working firing mechanisms. Stan was undaunted and built a firing mechanism that was far better than any of the others, shooting the balls further and recocking quicker.

But the ball firing mechanism is only half of a shooter robot; the other half is a mechanism to feed the balls into the barrel of the shooter. For this Stan ended up adopting a mechanism that was suggested to him by another team developing a shooter robot (he obtained their permission before developing his own version of their idea). The result was a robot that was quite competent at scooping up and firing balls—or so it seemed, when they operated the robot under direct human control.

When Stan and Dave approached the control problem, they made the assessment that the feedback control methods we were proposing for overall robot guidance were too error-prone and ambiguous. According to them, robots that used local feedback from the environment were inclined to wander and make missteps on their way to the feedback goal. Stan and Dave didn't trust local feedback as the most effective or reliable way to accomplish the task.

Instead, their approach was based on two unusual sensor applications. The first was a method to ascertain the robot's initial orientation on the playing table (*Robo-Pong* used a randomized angle of initial robot orientation). The second was a mechanism that was intended to keep track of the robot's position on the playing table in terms of a displacement from its initial position. Thus, by knowing the initial orientation and by recording all changes from it, they expected their robot to be able to navigate about the table at will. Surely, they believed, this would be a more effective approach than relying on erratic data sampled as the robot wandered about the playing surface.

It is worthwhile to examine these two sensor application strategies in detail, as both required innovation on their part and reveal the depth of their commitment to this approach.

The primary sensor provided in the *Robo-Pong* robot-building kit for detecting inclination was the mercury switch. However, the kit included an alternate inclination sensor, which we had not tested but could potentially operate more effectively than the mercury switches. The device was a small metal can, larger than a pencil eraser but smaller than a thimble, which contained a tiny metal ball. The can's base was flat and four wires protruded down from it. Inside the can, the four wires terminated in distinct contact areas. In operation, the ball rolled around inside the can and created an electrical contact between the inner wall of the can and one (or two) of the four contact areas. In a sense, the device

was a two-dimensional version of the rolling ball sensor (see Section 3.3), but without the discrete sensing problem.

Our reason for not relying on the metal can sensor as the primary inclination-sensing device was that we were unsure of how pronounced the contact bounce problem due to vibration would be—the difficulty that confounded participants in *King of the Mountain*. Because the sensors were quite cheap, we decided to provide them anyway, buying enough to give two to each *Robo-Pong* team in addition to four mercury switches. We left the testing and evaluation of the device to the participants.

It turned out that the bounce problem with the metal can was indeed troublesome compared to the mercury sensors', so most participants used the mercury switches. Stan and Dave, however, thought of an unusual use for the metal can sensors. Because the robots were to start the contest on the inclined surface, robots could use inclination sensors to determine their initial angle of approach up the slant of the incline. The bounce problem would be eliminated since the robot would be at rest at the start of the round.

Because the initial orientation was specified in thirty degree increments, there were in principle twelve discrete initial orientations. Stan and Dave realized that by using three metal can sensors (each of which gave an inclination reading accurate to one of four quadrants), rotated by thirty degrees with respect to one another, his robot could *precisely determine* its initial orientation. This precision would be necessary to combine with the other component of their strategy, a movement controller.

The movement controller was a method to enable the robot to keep precise track of its position as it navigated across the playing surface (finding its way from the starting position to the trough, for example). For this, Stan and Dave employed *shaft encoders*, a sensor technology for measuring the angular rotation of a shaft.

Shaft encoders are typically employed in mechanical devices in which angular travel or displacement is confined within limits and hence can be repeatably measured. For example, a shaft encoder on a joint of a revolute robot arm can measure the angle of the arm. Usually such encoders are used in conjunction with limit switches; to calibrate the joint for sensing, the controller will drive the joint until it triggers the limit switch, thereby determining the zero position of the joint. All subsequent readings are then made with respect to the zero

position, which can be reliably re-established at a later time.

For mobile applications, shaft encoders have questionable value in determining position, because of the problem of a vehicle's wheels slipping with respect to the surface on which it is moving, causing errors in estimations of where the vehicle has moved on the surface. In these applications, shaft encoders are best suited for velocity and approximate distance measures; the speedometer/odometer of an automobile is an example of this usage.

Stan and Dave were cognizant of the slippage problem, but believed they had a workaround solution that would yield acceptable results. Rather than attaching the shaft encoders to the drive wheels, which they did realize would be quite prone to slippage, they added free-spinning *trailer wheels* that would rotate when dragged along the driving surface by the robot's movement from the drive mechanism. They attached the shaft encoders to the trailer wheels, which presumably were much less prone to slippage than the drive wheels.

Stan described the plan to me during early stages of his implementation. I argued with him that the approach was dubious; mobile robotics researchers had built various robots that attempted to perform global positioning based on similar ideas, and they all had shortcomings that forced the robots to frequently recalibrate their estimates of position based on local environmental references. He was undeterred. Since I did not perceive my role as preventing him from exploring an idea that he was interested in, and there was no harm in his trying it, I backed off.

Stan and Dave proceeded with the concept; Dave performed much of the programming. At various points during the development of the robot they were able to make remarkable demonstrations of the robot's ability to respond to control commands and measure and correct for its motion. The robot could drive to specific commanded positions; or, if the robot were dragged along a table to a position eight or ten inches away and then released, the position controller would drive the robot back to within an inch of its original position. Of course this only worked when the robot was dragged with considerable downward pressure so that the trailer wheels tracked the movement (i.e., they did not slip). I tried to argue with Stan that as impressive as this demonstration was, the approach was still futile because displacements in game play would be of the sort that *would* cause slippage. I think at this point Stan was so pleased with his apparent success that he was not listening to me (I don't

believe I would have listened to such advice either, had I implemented a system so effective in laboratory testing).

The robot's overall strategy was to drive down into the bottom of its side of the table and sweep back and forth along the trough, firing balls over to the opponent's side. If the robot's initial orientation were such that it was aimed upward, it would drive forward to knock one of the three top balls over the edge, as an "insurance" measure, before retreating into its main activity in the trough.

During the testing, Stan and Dave maintained confidence in their design. Indeed, the robot seemed to largely work; often it would only require minor hand-administered nudges during practice rounds to be successful. In the actual contest, however, the design failed badly. The slightest deviation from ideal circumstances caused the robot to fail without possibility for recovery. In ignoring feedback from the specific features of the playing field, the robot really had no hope if any number of things were to go wrong: a slight miscalculation as to the amount of rotation needed to orient properly, a bump from the opponent robot, or a even an irregularity in the surface of the playing field. Stan and Dave had fooled themselves during the robot testing—when the safety net of their prodding corrections was gone, the robot could not reliably perform the sequence of steps needed to work, and it was a competitive failure.

Stan and Dave teamed up again to design a robot for the *Robo-Cup* contest. In his first written report for that project, Stan reflected on his experiences in building the *Robo-Pong* robot:

> **Lessons Learned.** Last year we spent a lot of time and resources on non-essentials. Our mechanical design was not stabilized until the night before the second contest. The majority of the time spent on software was in building a sophisticated object-oriented system. Our strategy was stable early enough, but it relied too heavily on unproven sensors and the accuracy of shaft encoders. This year we are focusing our energy on creating a robot that works well, even if we don't have time to make it look good.
>
> **Design Goals.** In the past, robots have often failed because one particular sensor is critical and the software doesn't handle exceptions and breakdowns very well. Our goal is to build a robot that can check for errors and compensate before the errors escalate. Reliability and robustness are the targets for our project. Given enough time for testing, most unexpected conditions can be

accounted for.

It seemed as if Stan had learned his lesson. Yet, remarkably, Stan and Dave proceeded to repeat exactly the same mistakes in their *Robo-Cup* robot design.

While Dave did not attempt to build as sophisticated a global positioning system, Stan and Dave did again employ shaft encoders as the primary sensor for their *Robo-Cup* design. In *Robo-Cup*, the contest task involved driving from the starting position to a ball dispenser, actuating the ball dispenser's control button (causing balls to drop from above), and delivering the balls to a soccer-like goal. Not surprisingly, Stan and Dave chose to shoot the balls in, reducing their robot's task to the job of successfully moving from the starting position to a location beneath the ball dispenser at which balls could be shot directly into the goal area.

Stan and Dave's insistence on using the shaft encoders to guide their robot's movement from the starting position to the ball dispenser caused them to ignore an obvious alternative solution: a reflectivity coding on the playing surface which would have given the robot the information it needed (i.e., where the ball dispenser is located).

In this year, one of their troubles was gone: rather than robots being subject to a random orientation at the start of the contest, they could be placed within the designated starting circle at any orientation by the robot's own designers. Knowing that an exact measure of initial position was critical to his robot's success, Stan requested permission to allow the use of a template that he could employ to position his robot repeatably at the same location! We consented.

Their robot's performance in the actual contest was disappointing. Stan and Dave were not able to tune the performance of the shaft encoders to be consistent. The night of the contest, the robot would stop its journey to the ball dispenser just about an inch short of the desired position, lock itself to the wall, and proceed to fire balls that would miss their target for the rest of the round. The location at which it would anchor itself was just at the threshold of being able to trigger the ball dispenser at all; in one round it was not even able to do so, and spent the bulk of the round in futile attempts to strike the panel that would dispense the balls.

There is a short coda to this story. Beginning with the *Robo-Puck* contest, we ran a rematch competition at the Boston Computer Museum each year, which took place several months after the date of the MIT contest. Stan rebuilt his *Robo-Cup* robot from the ground up for the rematch, this time using a strategy based on feedback from the playing field features rather than from shaft encoders. His efforts were rewarded: though Stan's robot didn't win the overall competition, it did win one round—Stan's first competitive victory in two years of robot-building.

4.3 Case Study Two: Simulation as a Development Tool

The first year that the Interactive C programming environment was available (the *Robo-Pong* class), there was a student who didn't want to use it. This student, "Tom," felt strongly that he would have a better learning experience if he programmed his robot in assembly language, thereby learning the "nuts and bolts" of the project hardware and the 6811 microprocessor, rather than being abstracted away from these things by the C language system.

Consistent with our approach of letting students delve into the aspects of the work that they individually found most interesting, we supported Tom in his endeavor. We did warn him that we would not have a great deal of time available to help him resolve various problems that he might encounter; this was agreeable to him as he specifically wanted to work through these sort of things himself. Tom was an accomplished MIT student in his junior year who had excelled in MIT's microprocessor architecture course and we felt comfortable that the arrangement would be productive.

Tom took a surprising approach to the project, however: he began by writing a program to simulate the operation of the 6811 microprocessor. His prior experience in the microprocessor architecture class had provided him with a powerful experience of the role of simulators. In the design optimization contest for the microprocessor course, Tom spent the bulk of his effort developing a compiler optimized for the microprocessor hardware,

rather than upgrading the hardware itself.[2] To assist in the development and testing of the compiler, he had created a simulator for the microprocessor hardware. This allowed him to test his compiler optimizations much more effectively than if he had to run its output on the target hardware directly.

Tom believed the same strategy would pay off in the Robot Design project. By creating a simulator for the robotic hardware, including the microprocessor, the robotic sensors, and the motors, he could develop much better final robot; he would not have to "muck around" with the robot hardware until he had a program that he had already tested to be effective on the simulator.

However, things did not go according to the plan. The job of creating a viable simulator for the robot project was more difficult than Tom expected. Aside from the issue of debugging the simulator itself—a simulator can do more harm than good if its own results are not trustworthy—there were aspects of developing the simulator that introduced unanticipated complexity.

One problem was related to complexity in the 6811 microprocessor itself. While Tom's simulator succeeded in being able to execute the basic stream of 6811 instructions, there were other aspects of the 6811 hardware that were more difficult to simulate (primarily, the interrupt facility). Unfortunately, these aspects could not be ignored; they were an intrinsic part of the operation of the electronic hardware attached to the 6811.

The other problem was more subtle and yet less tractable: the difficulty of simulating the microprocessor *in the context* of the actual robotic task. Here Tom was caught off-guard: the simulator for his prior microprocessor design project was relatively straightforward, because the entire system to be simulated lay in the realm of the microprocessor itself. The robotic system, however, was based on a microprocessor interfaced to physical sensors, motors, and mechanics. In order for the simulator to be a valuable tool, it would have to do the job of simulating input from the sensors and the effects of output from the motors—which meant that both the environment outside of the robot and the robot itself would have to be built into the simulation!

[2] This was an unusual approach; the format of the course encouraged students to improve the hardware to achieve better performance.

It so happens that this very approach is taken by some robotics researchers. Rather than dealing with the vagarities and expenses of developing physical robotic hardware, some researchers build complex robot simulations that they use to then study control algorithms, vision systems, path planners, and other aspects of robotics work.

Other robotic researchers are dubious or even scornful of the validity of this approach. To the "hands-on" roboticists, the value of any theory, model, or architecture of a robot control system is only as good as it can be tested on an actual robot with actual sensors, motors and computational hardware. By using simulators, this group claims, perplexing and deep questions are glossed over and the results obtained do not hold much validity. They point to studies developed on simulated experiments that break down when tested on actual hardware.

This tension between simulation versus reality is an on-going debate in the robotics field. Many in the simulation camp now include models of sensor noise and other physical problems—developed with actual experimentation—in their simulated environments. In other recent work, researchers test their control strategies on both physical robots and simulations, thereby gaining the virtues both: the irrefutability of an actual system and the versatility of a simulated one. An example is the work of Meeden et al. (Meeden, McGraw, & Blank, 1993).

In Tom's case, the results were disastrous from a performance point of view. By the time he realized the simulator was not going to be useful, much of the month had expired. When he finally delved into the actual programming on his robot, he progressively encountered a series of low-level hardware and software problems—ones which we had successfully shielded other students from in developing the Interactive C environment.

With literally a day to go, Tom gave up on the assembly language approach and attempted to get something working using Interactive C. It was too late, though; while he did field a robot on the eve of the contest, he had insufficient time to work through the higher-level control problems to get his robot to perform the task.

Even though the robot was a failure, this is not to say that Tom did not have a valuable learning experience. Certainly he learned in intimate detail a great deal about the 6811 microprocessor and the hardware we had developed around it, but there was a more profound

engineering lesson as well. I would guess that Tom would think twice about the complexity of a real-world system before pegging his hopes on a simulator again.

4.4 Analysis

The examples in this chapter illustrate that, rather than being "blank slates," students who participate in the Robot Design course have a variety of preconceptions about systems and control. These ideas are formed by experiences in the traditional academic curriculum, and warrant examination specifically because they are not particularly effective when applied to the Robot Design task.

In the first section of this chapter, *Introduction to Robotic Control*, we saw how many students have trouble seeing the contest task from a robot-centric perspective, and instead imagine the task from their own omniscient perspective. This naiveté was tempered when they tried to put their ideas into being and realized that the robot's point of view was very different from their own.

With few exceptions, all of the robots created for *Robo-Pong* and *Robo-Cup* used fundamentally algorithmic solutions. This fact is not surprising; there are at least two factors which would strongly encourage the formulation of algorithmic solutions.

One of these is the nature of the contests themselves, which require a series of activities to take place in a short period of time. A robot that wandered about as it waited to be guided into action by sensory stimuli would not be efficient; the typical winning robot employs a fast, reliable, and effective algorithmic strategy. If they're not interfered with, these robots successfully perform their task on nearly every run.

The other factor encouraging algorithmic solutions is the Interactive C programming language, which is fundamentally procedural. While it is a general-purpose programming language, and can in principle support the construction of a variety of control methodologies, as a procedural language it encourages students to create procedural control structures.

One feature of Interactive C, however, encourages reactive control: the multi-tasking capability. Indeed, this feature was used by students to implement reactive controls, as was discussed in the *Crazy Train* robot design. This evidence points to the effect that

the programming language has on students control ideas; extensions or revisions of the programming language could encourage students to think differently about control.

While it is then not unexpected that students create largely algorithmic robots, what *is* surprising is how poorly these systems perform with respect to the students' own expectations. Nearly all of the students who participate in the class are genuinely surprised by the difficulty of getting their robot to work reliably. They blame performance problems on failures of particular components of their systems, rather than re-evaluating overall approach to control.

The students' control ideas come from those presented in their university courses. At MIT, examples are found in the introductory Computer Science course, *Structure and Interpretation of Computer Programs* (course number 6.001), and the *Software Engineering Laboratory* course (course number 6.170). Among the central ideas developed in 6.001 is the concept of *abstraction*: that by encapsulating messy implementation detail into conceptual "black boxes," complex systems can be built from components with relatively clear functionality. Abstraction is a way of managing complexity—a way to keep minutia in check, and to create large systems with clearly understood parts. As stated in the text of the course (Abelson & Sussman, 1985):

> In our study of program design, we have seen that expert programmers control the complexity of their designs by using the same general techniques used by designers of all complex systems. They combine primitive elements to form compound objects, they abstract compound objects to form higher-level building blocks, and they preserve modularity by adopting appropriate large-scale views of system structure.[3]

The Software Engineering course expands and fleshes out this principle, teaching programming techniques like procedural and data abstraction through extended programming projects. The course makes a point of teaching how to write programs in a modular fashion, test code modules independently, and then integrate them into a complete system. Typical final projects are the design and implementation of a text editor or a computerized Othello game.

[3] *Structure and Interpretation of Computer Programs*, page 293.

While these projects are ideal for the goals of the course—teaching abstraction and modularity in the architecture of large computer programs—they promulgate the mindset that large systems are made from parts that are completely understandable and formally specifiable. In both of these examples, as in many others, the computer program consists of precisely formalizable data structures and algorithms to operate upon this known data to generate known results. Indeed, a large portion of the Software Engineering course is dedicated to methodology of testing with the intent of proving that a given program module will yield correct results when presented with data that satisfies particular conditions.

Much of the engineering curriculum is based on modern system theory: mathematical methods for understanding algorithms and systems that take particular inputs and yield particular outputs. Dynamical systems analysis, in which system state is represented by a state vector, and signal-processing theory based on transfer functions are leading examples. These domains of theory deal with systems that are perfectly represented in the mathematical models that students learn to manipulate.

Of course the value of such engineering knowledge is precisely that it allows us to construct systems that are indeed controlled to extreme degrees of precision. The great engineering successes are based on our ability to understand, model, analyze, and precisely control the world around us. But not all large systems are controllable by these means. As discussed by Ferguson, recent work on chaotic systems has challenged the assumptions of those working on automated traffic control systems:

> For engineers, a central discovery in the formal study of chaos is that a tiny change in the initial conditions of a dynamic system can result in a major unexpected departure from the calculated final conditions. It was long believed that a highly complex system, such as all automobile traffic in the United States, is in principle fully predictable and thus controllable. "Chaos" has proved this belief wrong. The idea that roads will be safe only when all cars are guided automatically by a control system is a typical but dangerous conceit of engineers who believe that full control of the physical world is possible.[4]

The example of the third section, *Simulation as a Development Tool*, raised the issue of the role of simulation in understanding and representing complex systems. For some

[4] *Engineering and the Mind's Eye*, page 172.

domains (e.g., VLSI design) simulation is a powerful and accurate technique for developing and testing complex systems. For others (e.g., structural engineering), there are difficult issues of determining the applicability of a simulated model to the actual artifact which affects the reliability of the simulation in profound ways. An additional complicating factor is that many simulation packages are developed by theoretical rather than practicing engineers, raising the question that the practicing engineer must trust the output of the simulation without necessarily knowing the founding assumptions upon which it is based. Henry Petroski quotes a Canadian structural engineer on this point:

> Because structural analysis and detailing programs are complex, the profession as a whole will use programs written by a few. These few will come from the ranks of the structural "analysts" . . . and not from the structural "designers." Generally speaking, it is difficult to envision a mechanism for ensuring that the products of such a person will display the experience and intuition of a competent designer.[5]

The reliance on simulation as an engineering technique is a consequence of assumption that complete control of the physical world can be attained. While simulations can reveal problems before they develop into serious engineering failures, over-reliance on the trustworthiness of simulations can lead to disasters. Contemporary examples of failures due to inadequacies in simulation abound; here are two striking ones from Peter Neumann's column in the *Communications of the Association for Computing Machinery* periodical (Neumann, 1993):

> On April 1, 1991, a Titan 4 upgraded rocket booster (SRB) blew up on the test-stand at Edwards Air Force Base. The program director noted that extensive 3-D computer simulations of the motor's firing dynamics did not reveal subtle factors that apparently contributed to failure. He added that full-scale testing was essential precisely because computer analyses cannot accurately predict all nuances of the rocket motor dynamics. (See *Aviation Week*, May 27, 1991 and Henry Spencer in *SEN 16*, 4, Oct. 1991.)
>
> The collapse of the Hartford Civic Center Coliseum 2.4-acre roof under heavy ice and snow on January 18, 1978 apparently resulted from the wrong model being selected for beam connection in the simulation program. *After* the collapse, the program was rerun with the correct model—and the results were

[5] *To Engineer is Human*, page 201.

> precisely what occurred. (Noted by Richard S. D'Ippolito in *SEN 11*, 5, Oct. 1986.)

If engineering students gain early hands-on experience with the complexity of electrical, mechanical, structural, and software systems, they will have more respect for the sorts of pitfalls that can be expected later when working on real projects, like in the case of the Civic Center cited above. With project experience like that in the Robot Design course, students learn some of the limitations of traditional control and analysis, and gain some of the experience that is needed to be a realistically-minded engineer.