Avoiding the Karel-the-Robot paradox: A framework for making sophisticated robotics accessible

Douglas Blank  
Computer Science  
Bryn Mawr College  
Bryn Mawr, PA 19010  
dblank@cs.brynmawr.edu

Holly Yanco  
Computer Science  
Univ. of Mass. Lowell  
Lowell, MA 01854  
holly@cs.uml.edu

Deepak Kumar  
Computer Science  
Bryn Mawr College  
Bryn Mawr, PA 19010  
dkumar@cs.brynmawr.edu

Lisa Meeden  
Computer Science  
Swarthmore College  
Swarthmore, PA 19081  
meeden@cs.swarthmore.edu

Abstract

As educators, we are often faced with the paradox of having to create simplified examples in order to demonstrate complicated ideas. The trick is in finding the right kinds of simplifications—ones that will scale up to the full range of possible complexities we eventually would like our students to tackle. In this paper, we argue that low-cost robots have been a useful first step, but are now becoming a dead-end because they do not allow our students to explore more sophisticated robotics methods. We suggest that it is time to shift our focus from low-cost robots to creating software tools with the right kinds of abstractions that will make it easier for our students to learn the fundamental issues relevant to robot programming. We describe a programming framework called Pyro which provides a set of abstractions that allows students to write platform-independent robot programs.

Introduction

The Karel-the-robot environment was designed to introduce structured imperative programming to beginning programming students (Richard E. Pattis 1981). In a similar way, inexpensive robots have made introductory AI topics accessible to a wide range of students, from K-12 to the college level. The availability of low-cost robots has led to their widespread use in the undergraduate artificial intelligence curriculum (Meeden 1996; Turner et al. 1996; Kumar & Meeden 1998; Beer, Chiel, & Drushel 1999; Harlan, Levine, & McClarigan 2001; Wolz 2001; Gallagher & Perretta 2002; Klassner 2002). Although this trend has been a tremendous help in bringing robotics to students, we believe these low-cost robot platforms often lead to a robotics dead-end, much the same way that over reliance on the Karel environment did to advanced programming paradigms. While low-cost robots, like the Karel environment, provide a wonderful motivation and a great starting point, the paradox is that they often trap the student in a single paradigm, or worse, a single hardware platform.

There are several problems with the use of low-cost robots in education. The first problem is that every robot platform comes with its own, often proprietary, development tools that are substantially different from other platforms. Often the primary programming languages used are different as well. More problematic may be a complete change in paradigm from one robot to another. Consequently, even if one were to invest in learning to use one robot platform, probably none of the code, and possibly little of the knowledge would transfer to a different platform. This situation is perhaps similar to the one in the early days of digital computers when every computer had a different architecture, a different assembly language, and even a different way of storing the most basic kinds of information.

Secondly, we believe that many robot programming paradigms do not easily support more sophisticated sensors. For example, low-cost robots often only come equipped with infrared range sensors. Some can be expanded to include sonar range sensors or even laser range sensors. However, we suspect that if even more sophisticated sensors were to become affordable, we would be unable to utilize them because there is no easy way of integrating them into existing robot software paradigms. That is, sophisticated sensors may be hardware accessible, but not conceptually accessible by the student. The problem is that the framework doesn’t pedagogically scale well.

We believe that the proliferated use of robots in AI education will result not from low-cost hardware platforms, but from accessibility of these platforms via common conceptual foundations that would make programming them uniform and consistent. Our position is defined more by striving for a lower learning cost of robotics without sacrificing the sophistication of advanced controllers.

Our goal is to reduce the cost of learning to program
robots by creating uniform conceptualizations that are independent of the size, weight, and shape of a robot. Consider writing a robot controller for obstacle avoidance that would work on a 24-inch diameter, 50-pound Pioneer2 robot as well as on a 2.5-inch diameter, 3-ounce Khepera. This was made feasible by making the following abstractions:

**Range Sensors:** Regardless of the kind of hardware used (IR, sonar, laser) sensors are categorized as **range sensors.** Sensors that provide range information can thus be abstracted and used in a control program.

**Robot Units:** Distance information provided by range sensors varies depending on the kind of sensors used. Some sensors provide specific range information, like distance to an obstacle in meters or millimeters. Others simply provide a numeric value where larger values correspond to open space and smaller values imply nearby obstacles. In our abstractions, in addition to the default units provided by the sensors, we have introduced a new measure, a **robot unit:** 1 robot unit is equivalent to the diameter of 1 robot, whatever it may be.

**Sensor Groups:** Robot morphologies (shapes) vary from robot to robot. This also affects the way sensors, especially range sensors, are placed on a robot’s body. Additionally, the number of sensors present also varies from platform to platform. For example, a Pioneer2 has 16 sonar range sensors while a Khepera has 8 IR range sensors. In order to relieve a programmer from the burden of keeping track of the number of sensors (and a unique numbering scheme), we have created **sensor groups** that can be abstracted and used in a control program.

**Motion Control:** Regardless of the kind of drive mechanism available on a robot, from a programmer’s perspective, a robot should be able to move forward, backward, turn, and/or perform a combination of these motions (like move forward while turning left). We have created three motion control abstractions: **translate**, **rotate**, and **move.** The latter subsumes both translate and rotate and can be used to specify a combination of translation and rotation. As in the case of range sensor abstractions, the values given to these commands are independent of the specific values expected by the actual motor drivers. A programmer only specifies values in a range -1.0..1.0 (see examples below).

**Services:** The abstractions presented above provide a basic, yet important functionality. We recognize that there can be several other devices that can be present...
from pyro.brain import Brain

class Avoid(Brain):
    def wander(self, minSide):
        robot = self.getRobot()
        #if approaching an obstacle on the left side, turn right
        if robot.get('range','value','front-left','minval') < minSide:
            robot.move(0,-0.3)
        #if approaching an obstacle on the right side, turn left
        elif robot.get('range','value','front-right','minval') < minSide:
            robot.move(0,0.3)
        #else go forward
        else:
            robot.move(0.5,0)
    def step(self):
        self.wander(1)

def INIT(engine):
    return Avoid('Avoid', engine)

Figure 1: An obstacle avoidance program in Pyro

on a robot: a gripper, a camera, etc. We have de-
vised a service abstraction to accommodate any new
deVICES or ad hoc programs that may be used in robot
control. For example, a camera can be accessed by
a service that enables access to the features of the
camera. Further, students can explore vision process-
ing by dynamically and interactively sequencing and
combining filters.

In the following section we explore an example that
utilizes these abstractions and demonstrates the effec-
tiveness of these abstractions in writing generic robot
controllers.

An Example

In this section, we’ll use the example of avoiding ob-
stacles to demonstrate the unified framework that Pyro
provides for using the same control program across
many different robot platforms.

Direct control is normally the first control method in-
troduced to students. It is the simplest approach be-
cause sensor values are used to directly affect motor
outputs. For example, the following pseudocode rep-

scribed in the previous section. It is written in an object-
oriented style, and creates a class called Avoid which
inherits from a Pyro class called Brain. Every Pyro
brain is expected to have a step method which is ex-
ecuted on every control cycle. The brain shown will
cause the robot to continually wander and avoid obsta-
cles until the program is terminated.

It is not important to understand all the details of
Pyro implementation, but the reader should notice that
the entire control program is independent of the kind of
robot and the kind of range sensor being used. The pro-
gram will avoid obstacles when they are within 1 robot
unit of the robot’s front left or front right range sensors,
regardless of the kind of robot.

After learning about direct control, students can
move to any of the other control paradigms. The
paradigms selected would depend upon the course that
Pyro was being used for. In a course that empha-
sized robotics, the next paradigm would most likely
be behavior-based control. An AI or machine learn-
ing course would likely skip behavior-based control and
move immediately to neural-network-based control.

Currently, the following modules are implemented
and extensive course-style materials are available: di-
rect control, sequencing control, behavior-based con-
trol, neural network-based learning and control, self-
organizing maps and other vector quantizing algo-
rithms, computer vision, evolutionary algorithms, and
multi-robot control. Other paradigms and modules are
planned in the future. These will include logic-based
reasoning and acting, classical planning, path planning
and navigation. Pyro is an open-source, free software
project, and we hope to get contributions from other interested users.

Conclusions

We have argued that it is more important to strive for easily learnable robot programming interfaces than for low-cost robot platforms. We have tried to avoid the Karel-the-robot paradox by carefully designing useful and universal conceptualizations. These conceptualizations not only make the robot programs more versatile, they also help in robotics research. Specifically, the modeling of robot behaviors can now be tested on several robot platforms without having to change the programs. This adds much credibility to the tested models as the results will have been confirmed on several robot platforms.

We believe that the current state-of-the-art in robot programming is analogous to the era of early digital computers when each manufacturer supported different architectures and programming languages. Regardless of whether a computer is connected to an ink-jet printer or a laser printer, a computer today is capable of printing on any printer device because device drivers are integrated into the system. Similarly, we ought to strive for integrated devices on robots. Obviously we’re not there yet. Our attempts at discovering useful abstractions are a first and promising step in this direction. We believe that discoveries of generic robot abstractions will, in the long run, lead to a much more widespread use of robots in education and will provide access to robots to an even wider range of students.

Acknowledgments

Pyro source code, documentation and tutorials are available at www.PyroRobotics.org. This work is funded in part by NSF CCLI Grant DUE 0231363.

References


