An Automatic Grader for Embedded Systems Projects

by Daniel Mendelsohn
MIT S.B., 2015

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

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

August 2017

Author .....

Department of Electrical Engineering and Computer Science
August 18, 2017

Certified by .....

Joseph Steinmeyer
Thesis Supervisor
August 18, 2017

Accepted by .....

Dr. Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

An Automatic Grader for Embedded Systems Projects

by

Daniel Mendelsohn

Submitted to the Department of Electrical Engineering and Computer Science
on August 18, 2017, in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

TODO: write this

Contents

1 Introduction

Write about 6.S08, the current situation, how its a pain, etc

Write about how "regular" software checkers are a solved problem

Write about unique challenges of embedded systems (computing platform challenges, physical vs digital abstractions, heavily time-dependent in a way typical software classes aren't)

2 Prior Work

Write about UT Austin's EdX class, and the system they use

Discuss strengths and weaknesses of the UT Austin approach

3 Design Principles

Before building MicroGrader, I outlined a set of design principles to serve as a foundation for my technical design.

3.1 Platform agnosticism

The microcontroller ecosystem is fractured and changes quickly. Unsurprisingly, a wide variety of platforms are currently used in embedded systems education. For example, MIT's 6.S08 uses the Teensy (a third-party Arduino variant) while UT Austin's Embedded Systems course on EdX uses the TM4C123 (an ARM-based board made by Texas Instruments). Ideally, the automated grader should not care if a project is implemented on a Teensy or a TM4C123, as long as it meets the appropriate specification. The architecture of the grader must not be intrinsically tied to any one microcontroller or any specific sensors. The interfaces ought to be as generic as possible.

3.2 Extensibility

It should be possible to define new data types for assessment. In 6.S08, students learn to work with a fairly standard set of sensors and other peripherals. The main system inputs are digital GPIO, analog GPIO, and a nine-axis IMU. The automatic grader should **not** be restricted to these types of inputs. For example, if a course assignment involves interfacing with a temperature sensor, the grader ought to be handle that.

This extensibility could even make it possible to use the grader in domains other than EECS. In principle, an embedded system provides an interface between the physical world and the digital world. MicroGrader could therefore be used to grade a project with any kind of electronically measurable output.

3.3 Actionable feedback for students

Students should be able to get explanations of each passed and failed test case. This may require significant technical work – the internal representation of a test case might be complicated, but the explanations should be simple. Based on the description of a failed test, a student should know exactly what output the system observed and what output the system expected to observe.

3.4 Ease of use for students

In terms of the student experience, MicroGrader should be seamless to use. Running tests should be as simple as setting a flag at the top of the program. Any embedded libraries that the students use should be configured to use that flag to determine whether to run normally or to run in "test" mode. Students should not have to reorganize or rewrite a functioning project in order to coax the grader into working properly.

3.5 Ease of use for course staff

My experience with tedious manual grading in 6.S08 inspired this project, in large part. If building test cases in MicroGrader is actually more time-intensive than manual grading, this project will have failed its primary purpose. The time required to customize the grader for a specific assignment should be roughly proportional to the complexity of the assignment. If possible, a working implementation of the assignment should be sufficient to programmatically generate a test.

MicroGrader should impose as few constraints as possible on the assignments it grades. In a perfect world, it would be able to evaluate if an embedded system implements any conceivable specification.

3.6 Minimize false positives

Although few students would complain about invalid solutions occasionally passing automated tests, it inevitably does them a disservice. In 6.S08, for example, the students are mostly newcomers to programming and have not yet developed sophisticated debugging skills or proper testing disciplines. Students have difficulty shedding the assumption that code is correct just because it passed the tests. If that code is critical in a future exercise, it will cause problems.

3.7 Minimize false negatives

It's discouraging and perplexing for students when apparently correct solutions fail automated tests. Ideally, any functional solution – not just the intended one – should pass the tests. It's a challenge to avoid "Heisenbugs", wherein the system works under normal operation, but fails during testing due to changes introduced by the test itself. In practice, I've found it necessary to develop a set of guidelines to avoid such situations (e.g. maximum recommended frequency for certain I/O events).

4 Defining a test case

We can think of a specification as a mapping from a set of input signals to a set of output signals. In this system, signals are assumed to be piecewise constant. A piecewise constant signal can approximate any piecewise continuous signal by increasing the resolution. Not all possible specifications can be evaluated by MicroGrader. For the sake of simplicity, the internal structures of MicroGrader only support a subset of all possible specifications, though this subset is quite broad.

4.1 Channels

We label each individual input and output as a *channel*. Specifically, a channel is comprised of a data type and a sub-channel. For example, "digital input" is a data type, the pin number is the sub-channel. The built-in data types, and their associated sub-channels, are:

- Digital Input (sub-channels are pin numbers)
- Digital Output (sub-channels are pin numbers)
- Analog Input (sub-channels are pin numbers)
- Analog Outputs (sub-channels are pin numbers)
- Accelerometer Input (sub-channels are axes)
- Gyroscope Input (sub-channels are axes)
- Magnetometer Input (sub-channels are axes)
- Monochrome Display Output (no sub-channels)

4.2 Static inputs

In a test case, the inputs signals are more-or-less statically pre-defined. This is in contrast to other automatic graders that deputize the student to perform actions that generate the

inputs. The details of how these pre-defined inputs are delivered to the student's program will be described later. As mentioned above, input signals are modeled as a piecewise constant function in time, with arbitrarily high time resolution.

Allowing for dynamic input generation at evaluation-time would greatly complicate the implementation and design of the system, and is outside the scope of this project. Typically, this option is chosen out of necessity, as other automated graders lack a way to deliver pre-define inputs. Dynamically generating inputs with human action is actually quite restrictive.

4.3 Evaluation of outputs

Outputs are evaluated "offline" in the algorithmic sense. That is, the outputs of the embedded system are collected in full, and then evaluated once the test is finished (for some definition of "finished"). Furthermore, for the sake of simplicity, MicroGrader considers each output channel independently. As implemented, there is no cross-modal evaluation of separate output channels.

Collected outputs are samples of continuous-time signal. We use the simplest method of mapping the discrete-time sampling to a continuous-time signal; at any given time, we consider the most recent sample of the output to be the current value of that output. This turns out to be pretty reasonable in the context of many embedded system outputs. On most microcontrollers, output values such as an electrical voltage are "held" until a new output value is specified.

4.3.1 Data structure: Evaluation Point

Our fundamental unit of output evaluation is the *evaluation point*. An evaluation point consists of:

- Output Channel (e.g. "analog output on pin 5")
- Time Interval (a numeric start time and end time)

- Expected Value (the "correct" value of the given output channel in this interval)
- Check Function (a boolean function of two arguments)
- Required Portion (a float between 0 and 1, inclusive)
- Condition (we'll discuss this later)

The check function takes an expected value and an observed value as input, and returns a boolean. We consider that boolean to be the correctness of the observed value with respect to the expected value.

In order to evaluate an evaluation point, we consider the reconstructed continuous signal for the point's channel, during the point's time interval. We then calculate the portion of the interval for which the observed value of the signal is correct, with respect to the point's expected value. If that portion is greater than or equal to the point's "required portion", then the point evaluates to `true`. Otherwise, it evaluates to `false`.

The point's time interval isn't an absolute time interval, but rather a relative time interval, where $t = 0$ is the time at which the point's "condition" is met. We'll discuss this is more detail in section 4.4.1.

4.3.2 Aggregating point results

For any channel, we calculate an overall test result in a flexible way. Specifically, an "aggregator function" is defined for each channel. This function takes the set of boolean point results for that channel as input, and returns a decimal score in the interval $[0, 1]$. The scores for each channel are then averaged together (optionally, a weighted average can be used) to get a final score for the entire test case.

4.4 Relative timing

What's wrong with absolute timing?

4.4.1 Data structure: Condition

Explain.

4.4.2 Relatively-specified inputs

How is a frame defined?

What if two frames overlap?

What value does the input have if no frame is defined?

4.4.3 Relatively-evaluated outputs

Quick explanation

5 Implementation

5.1 Architecture

How is the system organized?

Why?

What are the responsibilities of the client?

What does the host do?

5.2 Two-stage testing

Describe the interactive session

Describe the evaluation stage

5.3 Reporting results

What info do students get?

How does this interact with centralized courseware (e.g. EdX) [Hint: UT Austin's method is good]

5.4 Configurability

Describe what is configurable

5.5 Defaults

Describe how defaults make life easier

6 Limitations

[Intro sentence]

6.1 Latency

Write about latency (which limits frequency and inhibits tapping into I2C, SPI, etc)

6.2 Point-by-point evaluation

Explain how we cant do things like cross correlation

6.3 Step-wise continuity restriction

Explain how something like "check that integral of output is X" is incompatible with current software...but not ruled out by the overall approach.

7 Automatic Test Generation

Explain what this is and why it is helpful

Explain how we generate a recording (i.e. inputs are reported instead of requested)

Explain how recording becomes a test case...this will be a hefty section

8 Results

TODO: write this once I try out a lot of student 6.S08 code on MicroGrader

9 Future Work

TODO: write this at the end...when I actually know what I haven't done yet

"Derived output": where the output we care about is some time-dependent function of something the system produces (e.g. the system produces a motor voltage and we care about the position of the wheel driven by the motor)

10 Appendix

11 References