An Automatic Grader for Embedded Systems Courses

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 Courses

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

In many embedded systems projects, there are subsystems with unknown latency. Sometimes, this latency is highly variable. This is especially true in the context of MIT's 6.S08,

due to its focus on web-connected embedded systems. Events with highly variable latency include obtaining a WiFi connection, making an HTTP request, and obtaining a GPS fix.

In the current implementation of MicroGrader, web requests are not simulated. Rather, they are considered to be a hidden "internal" step between system inputs (e.g. a button is pressed) and system outputs (e.g. some text from the web is displayed). It is not possible, at the time a test case is designed, to predict when this output ought to occur in absolute terms. We can, however, specify when the output ought to occur in relative terms (e.g. when an HTTP response arrives). For this reason, MicroGrader typically represents time relative to the time at which a pre-defined condition is met.

4.4.1 Data structure: Condition

In MicroGrader, there are four basic types of conditions. Conditions and be composed together to create more complex conditions, which can represent most events we care about.

The first type of condition is the *basic condition*. The time at which a *basic condition* is met is defined by a *cause*. In this case, the cause is a boolean function of that takes a single input – a representation of an embedded system event. The boolean return value represents whether or not the input event is considered to satisfy the condition. For example, we can use a *cause* function that returns `true` if and only if the event is a "system initialization" event. Another possible *cause* function would return `true` if and only if the system event is a "Wifi response" event. Once the condition is satisfied, it remains satisfied forever. The details of how these events are observed will be discussed later in section 5.1.

The second type of condition is the *after condition*. This type of condition has another condition (of any kind) as its "precondition". The *cause* function of an *after condition* is only invoked for system events that occur after the precondition is met. In addition, the *cause* of an *after condition* can be a single number, rather than a function. In this case, we consider this condition to be met a fixed amount of time after the precondition is met, as defined by this number.

The third type of condition is the *and condition*. This type of condition has no *cause*, but rather just a set of *sub-conditions*, which can be of any type. This condition is considered to be met once **all** of the sub-conditions are met.

The fourth and final type of condition is the *or condition*. It is analogous to the *and condition*, except it is considered to be satisfied once **any** of its sub-conditions are met.

4.4.2 Data structure: Input Frames

Now that we have the ability to define time in a relative way, let's re-examine how a test case defines its inputs. Specifically, input signals are fixed relative to a condition, rather than be fixed at an absolute time. The principal data structure for accomplishing this is the *frame*.

A *frame* is comprised of a *start condition*, an *end condition*, and a set of time-series signals (each of which is associated with a specific input channel). The two conditions can be of any type described in section 4.4.1. The signals must all be defined for all times $t >= 0$. The time at which the start condition is met is considered to be our "anchor point", $t = 0$. A frame is considered *active* at times at or after the start condition is met, and before the end condition is met.

A test case can (and often requires) multiple frames, since different behaviors that we'd like to evaluate occur relative to different events. During the execution of a test, is it possible for zero frames, one frame, or multiple frames to be active at a given time. If exactly one frame is active, that frame's values for each input channel are the test inputs to the embedded system.

If multiple frames are active at a time t, we must decide which frame is used to determine the system inputs. To that end, each frame has a fixed integer *priority*. The active frame with the highest priority determines the system inputs.

If no frame is active at a time t, then depending on the configuration of the test case, default values can be used for each input channel, or an error can be thrown (ending the

test).

4.4.3 Relatively-evaluated outputs

Now, we can re-examine our Evaluation Point data structure. The point's time interval is a relative time interval, where $t = 0$ is the time at which the point's condition is met. This interval can only be translated to an absolute time interval at run-time, not the time the test case is defined.

5 Implementation

We have defined what a test case is a fundamental level (namely, a verification of an input-to-output mapping). We have also described MicroGrader's test case structure, which can represent a reasonable subset of all conceivable test cases. So far, we understand *what* MicroGrader does. This section details *how* MicroGrader does it. The implementation imposes some additional restrictions on the kinds of systems MicroGrader can evaluate. Most notably, the method in which MicroGrader "injects" pre-defined inputs to the embedded system makes it unsuitable for grading some types of projects.

5.1 Overall architecture

MicroGrader uses a client-server model. The server is not a web server, but rather a Python program running locally on a student's laptop or desktop machine. That program connects to the client – namely the embedded system being evaluated – via USB serial. The client is a thin wrapper of certain aspects of the embedded system's I/O functionality, which allows the server to "drive" system inputs and observe results.

This key design choice stemmed from a desire to change the execution of an embedded program as little as possible. A lightweight approach on the embedded side reduces the chances of incorrect evaluations (both false positives and false negatives).

5.2 MicroGrader client

The embedded client has two modes of operation. In *test* mode, we need to inject pre-defined inputs into the system instead of taking real input readings. The software, in lieu of taking a real reading, makes a request via USB serial to the server, which responds with the proper pre-defined input value. In the current reference implementation of the client, requests are blocking – the embedded system will simply wait for a response if it expects one. Currently, MicroGrader only supports one request "in-flight" at a time.

For system outputs, the client really does produce the output (e.g. a digital write, or displaying something on a screen). In addition to producing that output, the client reports the output to the server, for later evaluation. The client also reports certain other system events, such as the beginning of program execution and various Wifi activity.

The other mode of operation on the client-side is *inactive* mode. In this mode, the client makes no request to the server whatsoever. That is, the program ought to operate completely normally. Inputs are read from the actual sensors, and outputs are not reported. This mode allows students to experiment with an in-progress assignment without worrying about the grader.

The exact data that comprises client requests and reports varies depending on the type of request or report. The communication protocol is detailed in Appendix A. One commonality of all requests and reports is an integer timestamp, in milliseconds. The server will trust this timestamp, and doesn't keep track of its own time during a live test.

Ideally, the functionality of the MicroGrader client should wrapped in a library and should mostly be invisible to the student writing code "on top" of that library. It may also be necessary to modify existing libraries that handle I/O functionality. Details about the way this is accomplished in the reference implementation (for Teensy) can be found in Appendix B.

5.3 MicroGrader server

The MicroGrader server does the heavy lifting. It is responsible for interpreting a test case, and using that test case's information to respond promptly and correctly to requests from the client. We described the details of how a test case specifies input values in sections 4.2 and 4.4.2. The server is also responsible for observing the system outputs and grading them according to the test case's specification. The client has no knowledge of the whole test case, it only knows the server responses to its requests.

In the current implementation of the MicroGrader, all of the code is written in Python.

This language was chosen due to its ease of development and flexibility with data types. Importantly, functions are first-order objects in Python which makes the implementation of a test case much easier (after all, a test case includes aggregator functions and check functions). Python's popularity make it more convenient for others (especially time-strapped course professors and TAs) to understand and potentially extend the MicroGrader code base.

5.3.1 Two-stage testing

Evaluation of a student's system occurs via a two-stage process. First, the server and client engage in an interactive session to determine how the student's system responds. The second stage assesses these responses and determines an overall score.

In the interactive stage, the server loads the relevant test case and waits for the client to connect. Once the client connects, the server processes each message from the client and responds if necessary. Typically, only requests for input values require responses, though the client may request an acknowledgement response for all messages. For each incoming message, the server must update all *conditions* in all of the test case's *input frames*. This will allow the server to determine which frames are currently *active* and what response ought to be sent. A test case also includes an *end condition*. Once the end condition is met, the interactive session ends, the server disconnects from the client, and all activity during the session is recorded.

In the evaluation stage, the MicroGrader server considers all the activity that occurred during the interactive session. MicroGrader scans through the activity log in order to:

- Determine the time at which each *evaluation point's* condition was met, if it was met at all. This allows for a point's relative time interval to be converted to an absolute time interval.

- Reconstruct all the output signals. Recall that we assume that each new output holds until the next output on the same channel.

Each *evaluation point* is examined and evaluated. Refer back to section 4.3.1 for details. If an point's *condition* was never met, then that point automatically evaluates to `false`. The boolean results of each point are then aggregated into an overall score for that output channel. The scores for each output channel are averaged to generate a score for the test as a whole.

MicroGrader is **not** designed to gracefully handle errors. If something goes wrong with the communication between the server and the client, or the client sends an invalid request, the session ends immediately and the test fails with an error message. Similarly, if results of the session cannot be evaluated in the evaluation stage, the test automatically fails with an error message. These situations are typically indicative of a malformed test case or a software bug that is not the student's responsibility. Course staff should immediately address these types of issues.

5.3.2 Reporting results

For grading purposes, we only care about the numeric final score. A student, however, would be well served by examining a rich and readable description of the test results. In the event that a student does not receive full credit, this description should make it clear which *evaluation points* were `false` and why.

Specifically, for each *evaluation point*, the student can see:

- Whether that point passed, failed, or was not evaluated (which would occur only if a point's condition is not met).
- A text description of the point's time interval and the *condition* with respect to which that relative interval is defined.
- The output channel being examined.
- The expected value on that channel during the specified interval.

- A description of the check function being used to determine correctness.
- The portion of the interval for which the observed values were correct.
- A summary of the observed values on that channel in the specified interval. This summary is comprised of:
  - The unique values observed during the interval.
  - Whether or not each of those values was correct (as determined by the check function).
  - The portion of the interval for which each of those values was observed.

Some values do not lend themselves to a textual description. For example, screen images (a very common type of output), don't readily convert to text. To handle this, MicroGrader saves all screen images at PNG files in a special directory, and the text description of those images is just the file path.

For further readability, test results can be summarized somewhat more succinctly by only giving the full details for *evaluation points* that don't pass. Furthermore, if all *evaluation points* for a certain channel are correct, we can omit all the details for that channel. Students have access to both the full results and summarized results.

I considered a number of ways of reporting results to some online courseware (e.g. EdX, Coursera, etc.). In the end, I chose a simple but effective solution, inspired by UT Austin's Embedded Systems course on EdX. Students receive an individualized code number for each test through the online courseware, and enter that number into MicroGrader. When MicroGrader runs a test, the student's score is combined with that code and hashed. The student can then copy that hash back into the courseware. Since the number of possible scores is limited (only four digits or scoring precision are allowed), it's easy for the online courseware to determine the student's score. This solution is elegant in that it discourages cheating without requiring direct communication between MicroGrader and online courseware. To further discourage cheating, courses could require students to upload code.

5.3.3 Screen analysis

As previously mentioned, binary (i.e. no partial intensities) monochrome screens are a built-in type. MicroGrader also includes some utilities for working with such outputs, which are represented as a 2D array of binary values. Specifically, these utilities help instructors build sensible check functions for this type of value. After all, exact pixel-by-pixel screen matching is often not a reasonable requirement for students.

MicroGrader includes a built-in distance metric for binary monochrome screens. It's basic functionality is simple: for two screens of the same dimensions, the distance between them is the number of pixels that do not match. In practice, this fails to identify two screens that are very similar, but slightly shifted relative to one another. Therefore, MicroGrader's distance metric allows the user to specify a maximum allowable shift of one screen relative to another, in any direction. The distance is the **minimum** number of mismatched pixels. Essentially, the function tries all allowable shifts and picks the best one.

Many assignments involving a screen are text-based. That is, the screen is expected to contain printed text. To that end, MicroGrader includes functionality to extract the text from the screen. Various OCR engines were considered, but none worked acceptably well due to the tiny size of many fonts used in such contexts (potentially 5x7 pixels or smaller). Instead, MicroGrader looks for rectangles on the screen that match the exact bitmap of a character.

There are a few downsides to this text extraction approach. First, any noise in the image or overlying element makes extraction impossible. Furthermore, fonts must fixed-width. This isn't a fundamental constraint of this technique, but it does massively help performance since the algorithm only has to consider rectangles of a specific size. The biggest issue is that MicroGrader needs to know the bitmap for each character in the font. Since its sometimes not easy to find detailed information for embedded fonts, MicroGrader includes a utility to "record" the character bitmaps of a font using the actual screen output.

5.3.4 Configurability

MicroGrader was designed to be as configurable as possible. This manifests itself in a variety of ways.

Test cases themselves are highly configurable, as described earlier. After all, each *evaluation point* can have its own check function and its own condition to which its time interval is relative. Each point can also have its own threshold for the portion of the interval that must be correct in order for the point to evaluate to `true`.

Certain design decisions were made in order to make MicroGrader compatible with as many embedded platforms as possible. For example, it makes no assumptions about the resolution at which to discretize analog values. Any request for an analog input or report of an analog output comes packaged with information about the desired range and resolution of the analog quantity.

Later, it will be possible to add additional data types, beyond the built-in ones, without changing the source code. The easiest way to do this is by importing pickled [TODO: ref] Python objects that implement all the functions a data type needs to have. The minimum functions for a new data type are conversions to and from a byte encoding for transport, and a definition of the `==` operator.

5.3.5 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

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

9 Future Work

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

10 Appendix

10.1 Appendix A: Client-server protocol

Describe

10.2 Appendix B: Reference client implementation for Teensy

Describe

11 References