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

In this thesis, we introduce MicroGrader, an automated grader for embedded systems projects. The grader runs on a laptop or desktop computer, while the embedded system being evaluated runs on a microcontroller. By implementing a custom communication protocol between the grader and the embedded system, we enable the grader to inject test inputs and observe the resulting outputs.

We describe a specification format for instructors to define the technical requirements of an assignment. This format is meant to be simple to use, but highly expressive to allow for a wide range of possible assignments. We also outline the implementation of the MicroGrader system and the underlying communication protocol. We discuss the constraints that the specification format and the technical implementation impose on instructors. Finally, we describe a method to automatically generate specifications using a staff-built reference solution for a generic assignment.

Contents

1 Introduction

Verification of program behavior has been a major concern for engineers and academics since the early days of computing. The study of software testing has grown into a significant field of computer science. In this thesis, we will describe a software tool ("MicroGrader") that verifies the correctness of embedded systems, specifically in an educational context. Essentially, it is an automated grader for embedded systems assignments that provides actionable feedback to students.

We must understand why the verification of an embedded system requires a specialized approach, and understand how the educational context affects design choices.

First, let's discuss program verification in general, at a high level. In order to assess the correctness of any software system, we must first choose how to model such a system. Often, it is useful to model a software system as a finite state machine (FSM). That is, at any time, the output and next state of the system are uniquely defined by the input and current state of the system. Two FSMs are considered equivalent if they produce an identical sequences of outputs for the same sequence of inputs. If we only care about the functional correctness of a system, we should consider any FSMs that are equivalent to the correct "solution" to be correct as well. This implies we should only evaluate the output sequences and not the internal state.

While it is possible to exhaustively verify a black-box FSM, the number of steps required to do so is hyper-exponential, making this infeasible in practice. Rather, we typically test FSMs by verifying that they produce the correct output sequence for a useful subset of the possible input sequences. That useful subset of inputs is often chosen manually. The correct outputs might be defined manually as well. In the context of education, it is easier to use a "staff solution" to define the outputs. That is, the staff solution is canonically considered correct, and student solutions must match the output of the staff solution to be considered correct.

In most cases, the evaluator and the evaluated system can run on the same computing platform (e.g. an x86 processor with a compiler/interpreter for the relevant language). Typically, the evaluator exerts direct control over the evaluated system. For embedded systems, the resource constraints of a microcontroller – both in terms of memory and speed – force us to separate the the evaluator and the evaluated system. In the case of MicroGrader, the evaluator is a Python program running on a full-fledged processor, while the evaluated systems are embedded programs.

This separation of evaluator and the evaluatee (an embedded system in our particular case) is the primary focus of this paper, and it presents some new challenges. The evaluator requires the ability to specify the inputs to the embedded system, which are typically real-world sensor readings. Furthermore, it requires the ability to observe the outputs of the embedded system, which are typically physical output indicators (e.g. a motor, LED, etc). In MicroGrader, a communication protocol between the evaluator and the embedded system enables this access. The protocol is designed to be minimally intensive for the embedded system. Specifically, a low-level library on the embedded side implements one side of this protocol, which allows it to "request" inputs from the evaluator rather than performing actual sensor sampling. We can consider this an injection of test inputs. This library also allows for the embedded system to report outputs.

Beyond the separation of evaluator and evaluatee, there are additional differences between evaluating embedded and non-embedded software. Principal among these is the importance of timing. While the specification of a non-embedded program *could* include strict rules for the timing of specific operations, it's fairly unusual. Existing automatic graders may consider the overall program running time, but very few graders delve into the fine timing details. In embedded systems, timing is often of paramount importance. For example, in computing a single state update, an embedded system may require multiple inputs and produce multiple outputs. The inputs are not sampled at the exact same time, and the outputs are not produced at the exact same time. Furthermore, the implementation of the embedded program determines the timing of these events; an evaluator has no ability to compel the embedded system to sample an input at exactly time X or produce an output at exactly time Y.

Our evaluator cannot control the timing of input and output (I/O) events, nor can it ignore the timing of those events. This forces us to modify our “pure” FSM model, where I/O events occur at discrete time intervals. Rather than examining how an embedded system converts a set of discrete input sequences to a set of discrete output sequences, we must examine how it converts a set of continuous input signals to a set of continuous output signals.

The process of constructing test cases can be time-consuming. It is often prohibitively tedious for instructors to specify all the test inputs and expected outputs. Therefore, Micro-Grader includes a feature for “recording” the operation of an instructor’s system, and using that recording to programmatically generate test inputs and expected outputs.

2 Prior Work

In my search for existing automatic graders for embedded systems courses, I found only one system that's actually being used. The University of Texas at Austin (UT Austin) offers an EdX course "Embedded Systems - Shape The World" [1], which uses a grading system called TExaS. Most of what I learned about TExaS was gleaned by examining the course on EdX itself; I was not able to find technical documentation.

TExaS consists of a grader running on a student's computer, that observes the activity of the student's microcontroller. It is decidedly not multi-platform: the core of TExaS runs as a plug-in to the Keil Microcontroller Development Kit [2], which only runs on Windows. That plug-in works with the particular microcontroller used in the course.

The platform lock-in of TExaS has benefits. It appears to have low-level access to the operation of the microcontroller and places little – if any – burden on the microcontroller itself. TExaS can observe all the inputs and outputs of the microcontroller and it can even observe arrays of raw data in the microcontroller's memory.

TExaS does not inject test input values into the microcontroller. Rather, test cases prompt the student to perform certain operations (e.g. press button connected to pin 5). This reliance on interactivity and student control makes it difficult to construct complex test cases. Most of the assignments graded by TExaS are fairly simple from an I/O perspective: the main output is a small array of light-emitting diodes (LEDs). While I cannot speak to the usability on the instructor's side, the system seems to be limited in the scope of what it can do, but very capable of accomplishing grading tasks within that scope.

In designing and building MicroGrader, I wanted a cross-platform system to generalize the task of grading an embedded system. Furthermore, I wanted a broader set of possible inputs and outputs, and the ability to grade more complex tasks.

In MIT's 6.S08 (Interconnected Embedded Systems), microcontroller grading is com-

[1] https://www.edx.org/course/embedded-systems-shape-world-utaustinx-ut-6-10x

[2] http://www2.keil.com/mdk5/

pletely manual. Students show instructors solutions in-person or by video. This approach is time-consuming and does not scale well. 6.S08 uses CAT-SOOP [3] to evaluate certain software functions. Students enter C++ code, which is then evaluated on the centralized course server and evaluated within the CAT-SOOP environment. This allows the course staff to verify the student's code to some extent, but the execution of the code is inevitably different on the actual microcontroller. Students often find that solutions that work in the CAT-SOOP checker, fail on the microcontroller.

[3] https://catsoop.mit.edu/

3 Motivating example: "Wikipedia Scroller"

Before diving into technical details of the automated grader, it helps to have a motivating example in mind. MIT's 6.S08 includes an exercise called "Wikipedia Scroller", in which students build a digital encyclopedia using a microcontroller, a push-button switch, a nine-axis inertial measurement unit (IMU) and a 128x64 organic light-emitting diode (OLED) display.

Each student implements a tilt-based interface that allows the device's user to build an alphanumeric query string. Using the ESP8266 chip, the device transmits the query string to a server-side Python program. That Python program uses the query to formulate a query to the Wikipedia API, parses the response from the Wikipedia API, and transmits the first 200 characters of the relevant Wikipedia description to the embedded device.

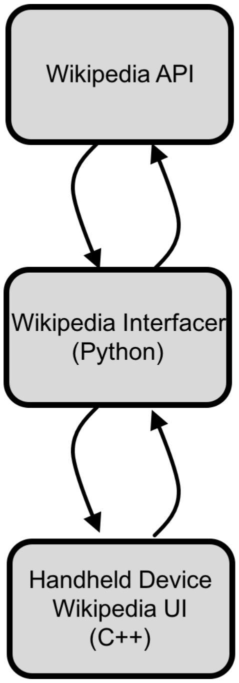


Figure 1: Block diagram for Wikipedia Scroller. Bottom block is the embedded system.

3.1 Specification for the embedded system

The embedded device allows its user to construct an alphanumeric query string, and it displays the results of the corresponding query. The primary input mechanism is “tilt”, as measured by accelerometer readings. In addition, the user can perform two actions using the push-button switch. We define a press to be a *short press* if its duration is under one second. We define a press to be a *long press* if its duration is at least one second. The system can be viewed as an FSM with three states. Note that, in the specifications for each state below, some of the finer details have been omitted in this paper for the sake of brevity.

3.1.1 State 0: display state

In this state, the result for the most recent query should be displayed. Specifically, the screen should show as many characters as possible from the most recent HTTP response. Initially, before any query has occurred, we define the “most recent result” to be the empty string. The system’s built-in 5x7 fixed-size font should be used, so that eight lines, each comprised of 21 characters, can fit on the screen. See Figure 2 for an example. When the user performs a long press, the system should switch to the query entry state.

TODO: replace Figure 2 with a better photo, without html tag

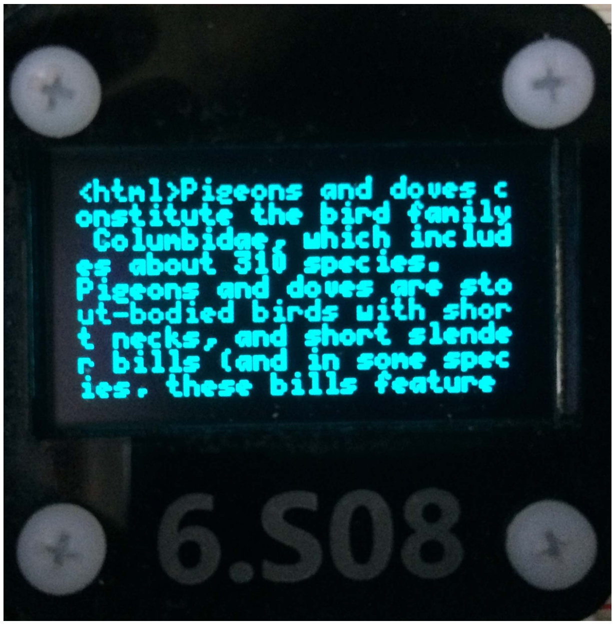
3.1.2 State 1: query entry state

In the query entry state, there are two relevant variables: `query` and `next_char_index`. Initially, `query` is the empty string, and `next_char_index` is 0. By definition, we say that the “next character” is the character in the string

`"abcdefghijklmnopqrstuvwxyz0123456789"`

at position `next_char_index`. So, the next character is initially `"a"`.

In order to build the query string, the user selects characters one by one. The user tilts the device left and right to “scroll” through the possible options for the next character. If the device is tilted 20° to the right, then `next_char_index` should increment every 150



```
<html>Pigeons and doves c
onstitute the bird family
 Columbidae, which includ
es about 310 species.
Pigeons and doves are sto
ut-bodied birds with shor
t necks, and short slende
r bills (and in some spec
ies. these bills feature
```

Figure 2: Screen output when in the display state. The last query was “pigeons”.

milliseconds. If the device is tilted 20° to the left, then `next_char_index` should decrement every 150 milliseconds. The index should wrap around, so that `"9"` and `"a"` are essentially adjacent characters.

With a short press, the user “locks in” the next character. That is, the next character should be appended to the query string, and then `next_char_index` should be reset to 0. Finally, with a long press, the user finishes the construction of the query string, which is then transmitted via HTTP to the server-side portion of the project.

In this state, the text on the OLED screen should always be `query`, plus the next character. For example, if `query` is `"pigeo"` and `next_char_index` is 0, then the string `"pigeoa"` should be displayed on the screen.

TODO: add some demonstration photos

3.1.3 State 2: waiting state

In this state, the device waits for a response from the server. When the response arrives, the system should transition back to the display state (which is responsible for displaying the response). In this state, the OLED screen should contain the text `"Waiting for response"`.

3.2 Manually grading Wikipedia Scroller

As a teaching assistant (TA) grading 6.S08 assignments, the author performed the following procedure to check that the assignment was implemented correctly.

First, upon system initialization, he checked that the OLED screen was blank. Then, he performed a long press to transition to the query entry state and verified that the character `"a"` appeared on the screen. He then tilted the device slightly left and right (less than 20°) to ensure the text did not change, signifying that the scroller was not too sensitive to tilt. He would then increase the tilt to more than 20° to ensure that scrolling occurred. When holding a constant tilt at a high angle, he checked the rate at which the character changed. He also checked that wrap-around was correctly implemented in both directions (i.e. that a left tilt from `"a"` yielded `"9"` and that a right tilt from `"9"` yielded `"a"`.

After verifying that the scrolling behavior was correct, he entered a short query string to ensure that string-building worked. He performed a long press to send the query to the server and checked that the proper response appeared on the screen shortly thereafter. Finally, he performed a long press to prompt a transition to query entry mode again, and verified that the old query had been cleared and the screen showed just `"a"`.

In this manner, the author was able to grade each student's work in under two minutes. In a class of 180 students, six hours of grading time was required for this assignment alone. Furthermore, it was difficult to give students detailed feedback, and students only received feedback after it was too late to fix any mistakes. An automated grader would solve these problems.

4 Defining a test

In this chapter, we describe the design of the *test case* data structure, and explain the reasoning behind it. At an abstract level, a grader's role is to assign a score to an embedded system. A human grader might rely on intuition or a checklist in order to assign a score. For MicroGrader, the test case defines the process by which a score is assigned. A MicroGrader test case serves two primary functions. First, it specifies any necessary input signal signals. Second, it specifies how the resulting output signals should be graded.

4.1 The shortcomings of statically defined inputs

While developing MicroGrader, we first explored a static approach to specifying inputs. In this design, all input signals would be completely predefined. For example, a test case could specify a digital input as seen in Figure 3.

Figure 3: TODO: Figure and caption of simple digital signal

In this particular example, presuming the digital input is connected to an active-low push-button switch, the test case indicates the button is pressed in the interval $(0, 0)$ and $(0, 0)$ (TODO: fill in values) and not pressed at all other times. We would define $t = 0$ as the time at which the microcontroller last powered on or reset. In this static paradigm, the input signals would not depend upon the behavior of the embedded system.

In practice, we found that statically defined inputs limited scope of assignments for which MicroGrader could be used. Grading assignments often requires input signals that are dependent on the behavior of the system. Consider "Wikipedia Scroller" for a concrete example. In particular, Wikipedia Scroller requires that the system wait for an HTTP response after transmitting a query. The length of time spent waiting for the response varies widely and is not within the student's control. When human graders evaluate the system, they know to wait for the HTTP response, whether it takes one second or five seconds, before performing additional test actions. We should consider the timing of subsequent actions relative to the arrival of the HTTP response, not relative to the initialization of the system as a whole. We can conclude that, in order to effectively grade Wikipedia Scroller, MicroGrader test cases need to dynamically define inputs, with respect to run-time events.

4.2 Dynamically defined inputs

TODO: introduce concept of "primitive event". But we need to describe unique events to make relative time make sense...

4.2.1 Data structure: "Condition"

In order to specify unique run-time events, we must first introduce a data structure that describes those events. We will refer to this data structure as a *condition*. As an embedded system executes a program, MicroGrader can determine the condition's $t_{\text{satisfied}}$, the elapsed time between the embedded system's initialization and the condition's satisfaction. A condition's $t_{\text{satisfied}}$ can be undefined, signifying that the condition was never satisfied during the embedded system's execution.

Our condition data structure can express unique events such as: "The first HTTP request", "The second HTTP request", and "The first button press after the second HTTP response". It achieves this with a recursive design; complex conditions can be built by composing simpler ones. There are four fundamental types of conditions.

The first type of condition is the *basic condition*. A basic condition's $t_{\text{satisfied}}$ is defined by a *cause*. A cause is boolean function that takes a single input representing a MicroGrader primitive event. As MicroGrader processes each primitive event during the execution of an embedded program, the cause function is invoked with that primitive event as its input. The condition's $t_{\text{satisfied}}$ is the **first** time at which the cause function returns `true`. TODO: give "first HTTP request" example

The second type of condition is the *after condition*. After conditions have another condition (of any kind) as a “precondition”. Like basic conditions, after conditions have a cause function. This cause function is only invoked for primitive events that occur after the precondition is satisfied. As before, the after condition’s $t_{\text{satisfied}}$ is the first time at which the cause function returns `true`. TODO: give “second HTTP request” example

Alternatively, the cause of an after condition can be a single number, t_{cause}, instead of a function. In this case, we consider this condition’s $t_{\text{satisfied}}$ to be the precondition’s $t_{\text{satisfied}}$ plus t_{cause}. In general, an after condition cannot be satisfied without its precondition being satisfied.

The third type of condition is the *and condition.* This type of condition has no *cause*, but rather a set of subconditions, which can be of any condition type. This condition is considered to be met once **all** of the subconditions are met. Equivalently, this condition’s $t_{\text{satisfied}}$ is the maximum of its subconditions’ $t_{\text{satisfied}}$ values, or undefined if any subcondition has an undefined $t_{\text{satisfied}}$.

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. Equivalently, this condition’s $t_{\text{satisfied}}$ is the minimum of its subconditions’ $t_{\text{satisfied}}$ values, or undefined if all subconditions have an undefined $t_{\text{satisfied}}$.

TODO: Give a big example with a sequence of primitive events and various conditions

4.2.2 Data structure: “Input Frame”

TODO: introduce InputFrame, together with examples of the two inputs frames for Wikipedia Scroller. Use images! Fundamentally, input frame is function: (channel, t) -¿ value. Channel is a label that specifies a signal (give relevant examples). Values can have different forms depending on type (e.g. digital vs analog)

4.2.3 Determining input values

TODO: how are input frames used?

TODO: what about times when 0 or 2+ frames are active? What then?

4.3 Grading system outputs

TODO: like old version, but better visual presentation of eval point data structure, and a worked example using Wikipedia Scroller (second frame is easier)

4.4 Complete test case for Wikipedia Scroller

TODO: fill this in, with pretty formatting

5 Defining a test case (old)

We can think of an embedded system as a mapping from a set of input signals to a set of output signals. We define a specification to be a set of permissible mappings. In MicroGrader, signals are assumed to be piecewise constant. A piecewise constant signal can approximate any piecewise continuous signal to arbitrary precision by increasing the resolution sufficiently.

5.1 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 Output (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)

5.2 Static inputs

In a test case, the inputs signals are more-or-less statically predefined. This is in contrast to UT Austin’s TExaS grader, which deputizes students to perform physical actions that generate the inputs. For example, if a TExaS test case calls for a button press, the system would prompt students to physically press the button themselves. MicroGrader can specify inputs without student action. 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 human-driven input generation 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 predefined inputs. Dynamically generating inputs with human action is actually quite restrictive.

5.3 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. Currently, there is no cross-modal evaluation of separate output channels.

Collected outputs are samples of a continuous-time signal. We use the discrete-time sampling to reconstruct an approximation of the continuous-time signal. Specifically, 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 reasonable in the context of many embedded system outputs. Often, microcontrollers hold output values constant until a new output value is specified.

5.3.1 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 will 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. This is described in more detail in section 5.4.1.

5.3.2 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 numeric 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.

5.4 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 internet-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, internet 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 internet is displayed). It is not possible, at the time a test case is designed, to predict when outputs should occur in absolute terms, because variable latency events cause unpredictable timing. We can, however, specify when the output should occur in relative terms (e.g. relative to the arrival of an HTTP response). For this reason, MicroGrader typically represents time relative to the time at which some condition is met.

5.4.1 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 7.1.

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.

5.4.2 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 5.4.1. 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).

5.4.3 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.

5.5 Example test case for Wikipedia Scroller

TODO: fill this in

6 Design principles

TODO: merge this chapter and the ”Implementation” chapter into one called ”Technical Design”

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

6.1 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). UT Austin's Embedded Systems course on EdX uses the TM4C123, a Texas Instruments microcontroller with an Advanced RISC Machine (ARM) instruction set. [TODO: add reference] 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 should be as generic as possible.

6.2 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 general-purpose input/output (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 should 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.

6.3 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.

6.4 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.

6.5 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. It should be able to evaluate a broad range of technical specifications.

6.6 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.

6.7 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.

7 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.

7.1 Overall architecture

MicroGrader uses a client-server model. The server is not a internet 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).

7.2 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 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 should 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.

7.3 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 5.2 and 5.4.2. 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.

7.3.1 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 should 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 5.3.1 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.

7.3.2 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 image 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.

7.3.3 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. Yet another built-in function checks that if a screen is significantly closer than a blank screen to a given reference screen. After all, a blank screen might match a large number of pixels but isn't very close in terms of informational content.

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.

7.3.4 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 serialized [4] 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 equivalence operator.

7.3.5 Defaults

The high level of configurability could potentially make it difficult for an instructor to use MicroGrader. Therefore, wherever possible, reasonable defaults are included. For example, the default aggregator function is "portion of `true` values" (recall, an aggregator converts a set of boolean evaluation point results into a numeric channel score). The default check

[4] https://docs.python.org/3/library/pickle.html

function is the equivalence operator. Each built-in data type has a default value.

Furthermore, defaults make it easy for instructors to dynamically construct a test case using a working implementation of assigned embedded project. We'll discuss that more in section 9.

8 Limitations

Although MicroGrader is well-suited for grading many types of microcontroller projects, there are limitations that any instructor using this tool should know about.

8.1 Latency

In order to implement its side of the protocol, embedded clients must do some work. Effort has been made to minimize this, but this issue nevertheless makes MicroGrader unusable for projects that have very sensitive timing, or that involve performing I/O operations at high frequencies. In *test* mode, the latency of reading an input is often orders of magnitude higher than in *inactive* mode. The embedded client must wait for the MicroGrader server to process and respond to a request for an input value. Most of the latency is fundamentally due to the round-trip latency of the USB protocol, and due to the firmware on the server machine.

The latency issue makes is impossible for MicroGrader to examine raw communication channels such as UART, I2C, or SPI, or to inject values into those channels at a reasonable rate.

The specific constraints depend on the platform. In building the reference implementation for Teensy, I observed that round-trip latency for relatively small messages (< 100 bytes) was about a millisecond. Therefore, the upper bound for the input sampling rate is around 1 KHz. This is fast enough for many assignments that involve “human” time scales, but woefully insufficient in other cases, such as audio sampling.

8.2 Point-by-point evaluation

Our *evaluation point* model inherently forces a test case to independently evaluate points of an output signal. For any given interval, there is a single expected value. More sophisticated correctness metric, involving the entire output signal at once, aren’t possible in this framework. For example, we cannot use *evaluation points* to determine the cross-correlation between the observed output signal and an expected output signal.

8.3 Step-wise constant restriction

MicroGrader assumes that the embedded system's outputs remain constant until the next output is observed. While a step-wise constant function can approximate any step-wise continuous function to arbitrary precision, this can put an unacceptably high burden on the embedded client. For example, both Arduino and Teensy have built-in functionality for creating square wave signals of a specified frequency for a specified duration, using a single user-facing instruction (`tone(pin, frequency)` or `tone(pin, frequency, duration)`). In order for the embedded client to properly inform the server about the square wave signal, it would have to send a message on every rising and falling edge. It is not possible, in a single message, to inform the server about the square wave that is being produced.

8.4 Derived outputs

In many systems, the embedded device produces outputs that serve as inputs to some physical plant, which then produces the outputs we actually care about. For example, a microcontroller might drive the voltage across the terminals of a motor, connected to a wheel. Perhaps we care only about the position of the wheel, not the motor's terminal voltage. Even under the assumption that the motor's angular velocity is proportional to the terminal voltage, and the motor responds instantaneously, we would have to integrate the microcontroller's output to derive the pertinent physical output.

It is currently impossible to evaluate derived outputs like this using MicroGrader (unless the derived output is zeroth order linear function of the microcontroller's output). In a later version, MicroGrader will include the ability to add a custom physical model to convert the raw output into a derived output.

9 Automatic Test Generation

Using CAT-SOOP as a TA for MIT's 6.S08, the process of building test cases is a quick and convenient one. An important convenience is the ability to only specify inputs for test cases. CAT-SOOP uses a staff solution to generate the correct outputs for those specified inputs. MicroGrader takes this philosophy one step further: it is possible to automatically generate a test case, including the desired system inputs, using a functioning staff solution.

9.1 Re-examining Wikipedia Scroller

Let's refer back to the Wikipedia Scroller exercise described in section 3. MicroGrader is capable of grading this assignment in a manner similar to a human grader. It injects a representative sequence of inputs, and then checks to see that each expected output is observed on the screen at the correct time.

It would be fairly tedious, however, to manually construct a test case for this with MicroGrader. An instructor would have to specify the entirety of the accelerometer input signal with exact timestamps, as well as the button input with exact timestamps. Both of these inputs change values about a dozen times.

Constructing the evaluation points would be arduous as well. The OLED changes at least 28 times in the above example. While not all of these necessarily require checking, many of them do. Even with MicroGrader's text extraction feature, it took quite a while for me to build a reasonable set of evaluation points. Clearly, this process could use some further automation.

9.2 Recording a reference solution

In order to obtain a recording a correctly implemented solution, we use introduce a third client-side mode, which we'll call *recording* mode (the first two modes were *test* and *inactive* modes).

Recording mode is quite similar to test mode, with one key difference. In test mode, input sampling is replaced with requests to the MicroGrader server. In recording mode, by contrast, input sampling occurs normally (i.e. sensors are actually used). Those real-world values are then reported to the MicroGrader server.

An instructor with a working solution can switch the client to recording mode, perform some set of tasks that fully demonstrate the functionality of the system. Meanwhile, the MicroGrader server makes a full record of this activity, including both inputs observed and outputs generated. That activity log is then used to programmatically construct a test case.

9.3 Constructing a test case

Completely generating a test case from a log along turns out to produce poor results. There are plenty of ambiguities in terms of how the instructor might want to the assignment to be graded. MicroGrader uses a lightweight data structure called the *scaffold* to allow for instructors to specify these preferences. The scaffold and activity log, when combined, can be used to produce a full test case.

9.3.1 Data structure: Scaffold

The two primary aspects of the *scaffold* are *frame templates* and *point templates.*

Frame templates arise out of the need to specify which intervals of time are relevant from a grading standpoint. Specifically, the generated test case needs to know when to "care" about the system outputs, and when to ignore them (e.g. while waiting for a internet request to resolve, or while waiting to get a GPS fix). The relevant intervals might comprise only a small portion of the total operation of the system.

The most important two components of a frame template are the *start condition* and *end condition.* These are instances of our familiar *condition* data structure, and will serve as the start and end conditions for the frame that will be generated. Refer back to section 5.4.2 for more information.

In our Wikipedia Scroller exercise, we can use a scaffold with two frame templates for the evaluation procedure previously outlined. The first frame could start when system initialization is complete, and end when the first Wifi request occurs. The second frame could start when the first Wifi response arrives, and end when the recording ends (I usually end recordings by just unplugging the client system). This two-frame structure allows the test case to ignore the variable latency period of time when the Wifi request is awaiting a response. After all, the length of this time period shouldn't be relevant in determining the correctness of a student's solution. If we tried to use one frame for this whole evaluation, the student solution's timing after the variable-latency event would certainly mismatch the staff solution's timing.

The other major aspect of a scaffold is the set of *point templates*. For each output channel, a point template is defined. This template will indicate how the instructor wants each *evaluation point* to be configured.

The three components of a point template are:

- Check Function
- Required Portion
- Time Interval

Using these point templates, we can specify aspects we want all evaluation points (for a certain channel) to share. For example, an instructor might want a special check function for values on the *Screen* channel. Perhaps in the case of our Wikipedia Scroller, that check function would verify that the text of the expected value and observed value are the same (regardless of where on the screen that text occurs). Furthermore, an instructor might want to impose a strict evaluation of a specification, requiring that student's solutions match the staff solution for a high portion of the time interval. On the other hand, an instructor may opt for a looser evaluation and reduce the required portion.

The time interval component of the point template is somewhat difficult to describe without a worked example. We'll get into that in section 9.3.3.

9.3.2 Constructing test inputs

We construct a full set of input signals for each frame. First, the entire activity log is scanned to determined the times at which the start and end condition for each frame template were fulfilled. We only generate a frame from a frame template if both the start and end condition were met, and the start condition was met first.

For each generated frame, we consider all input values recorded by the client between the times that the start and end condition were met. We convert the absolute times of these recorded inputs to relative times by subtracting the time at which the start condition was met.

One issue we haven't addressed yet is interpolating the recorded inputs. After all, those represent a sample of a continuous signal. MicroGrader gives instructors several options as to how to interpolate the sampled input – the following methods are all supported. For the sake of clarity, consider a value recorded at time t_n. The previous recording occurred at time t_{n-1} and the subsequent recording occurred at time t_{n+1}.

- Start: a recorded value holds in the interval $[t_n, t_{n+1})$
- End: a recorded value holds in the interval $(t_{n-1}, t_n]$
- Mid: a recorded value holds in the interval $[(t_{n-1} + t_n)/2, (t_n + t_{n+1})/2)$
- Linear: perform linear interpolation with a specified resolution between each sample. Note that this doesn't work for certain input types (namely digital inputs).

9.3.3 Constructing evaluation points

Just as with input generation, we first determine which frame templates generate a frame by determining the times at which their conditions were fulfilled. Again, for each frame, we consider the outputs on all channels that were reported between the fulfillment of the frame's start and end conditions.

For each (frame, channel) pair, we create a set of evaluation points. Specifically, we create a new evaluation point each time the output value changes on that channel. It is important to note that repeated output reports of the same value have no effect. For example, if the voltage on a pin is currently 0, and the microcontroller executes a command that sets the voltage on that pin to 0 again, that is functionally equivalent to doing nothing. The change in the value is all that matters.

In constructing each evaluation point, we can easily determine the channel. The expected value is equal to the observed value, since this is a staff recording and we consider its outputs to be canonically correct. For the point's check function and required portion, we use the values for the point template for this channel. For the point's condition, we use the start condition of the frame we are currently considering.

To complete our construction of an evaluation point, we need to specify the time interval. The time interval is generated using a combination of the time interval of this channel's point template, and the activity log. Specifically, the template time interval is not relative to the fulfillment of the frame's start condition, but rather relative to the time at which the output was reported. So if the template time interval is `(0, 100)`, the frame's start condition was met at $t = 1000$, and a new output is reported at $t = 1500$, then the evaluation point for this output will have an interval of `(500, 600)`. Recall that an evaluation point's time interval is relative to the point's condition being met. More generally, if the template interval is (t_{start}, t_{end}), the condition is met at t_{cond} and the output is reported at t_{output}, the evaluation point's interval will be $(t_{output} - t_{cond} + t_{start}, t_{output} - t_{cond} + t_{end})$.

A fixed-width time interval, like the one mentioned in the previous paragraph, is not very useful in a template. Typically, the length of the interval should be relative to the length of time for which the output was sustained in the correct solution. A fixed interval of width 100 milliseconds would be unreasonable if the correct solution sustained that output for only

50 milliseconds. Furthermore, it would be too lenient if the correct solution sustained that output for 5000 milliseconds.

To give instructors the ability to tune the width of evaluation intervals in a dynamic way, elements of the template time interval can be mathematical formulae (represented in string form) rather than just integers. In particular, this string can include a special variable, `T`, representing the length of time for which the recorded output was sustained. For example, a point template might have the time interval `("0.2*T", "0.8*T")`, which indicates that the evaluation point's interval should cover the middle 60% of the observed interval.

9.4 Auto-generated test case for Wikipedia Scroller

For an instructor to automatically generate a test case, he or she would first have to record a correctly implemented system performing the relevant operations (in our example, creating the query string "cat" and fetching the relevant information from the internet). For this assignment, a fairly simple scaffold would do. Consider the following possible scaffold:

The scaffold has has templates specifying the beginning and end of two separate frames. The first frame should begin upon system initialization and end upon the first internet request. The second frame should begin upon the response to that internet request, and end a few seconds afterward. For point templates, the default point template suffices, with a time of interval of `("0.2*T", "0.8*T")` as mentioned above. This gives a bit of leniency on the exact timing of state transitions in the student's solution.

With this simple scaffold, and the recording of the staff solution, MicroGrader can generate a test case that checks for all the outputs that a human grader would look for. Furthermore, the test case's use of two separate frames ensures that students would not be penalized for internet requests that are quicker or slower than the one in the original recording.

9.5 Resolving ambiguous cases

There is a major potential problem when recording a canonically correct staff solution. First, the reconstruction of the input signals from the recorded input samples is inevitably imperfect. Staff members should keep this in mind designing their solutions. Specifically, it's a good idea for the staff solution to sample (and therefore report) the input at a relatively high frequency even if such a high frequency isn't necessary for a correct implementation. This will give the MicroGrader server a maximally high resolution view of the inputs that the staff member physically created.

Even if we could, somehow, record an infinitely high resolution input signal, some recordings could have problems. Namely, the desired system output could be ambiguous for that particular input signal. This commonly happens in edge cases. It is possible to make the specification stricter to avoid such ambiguities, but that might be undesirable from a pedagogical standpoint. The instructor's solution would record one acceptable output. As a result, other acceptable outputs would be seen as incorrect.

A subsequent version of MicroGrader will include tools for instructors to avoid ambiguous inputs. Specifically, these tools will allow instructors to edit recordings of input signals. Presumably, an instructor would be able to remove any ambiguities. After editing, these new input signals would be injected back into the instructor's embedded system to record the unambiguously correct output.

10 Future Work

This thesis is being submitted with the described functionality, but many of the aforementioned shortcoming can be overcome. A version 2.0 of MicroGrader is under construction, and it will include the following improvements.

10.1 Custom data types

In the current implementation of MicroGrader, adding custom data types requires modifying the source code. In the next version, it will be possible to specify a custom data type in a separate file. The file will include the name of the data type, as well as the implementations (in Python) of certain necessary functions. Namely, there must be functions to serialize and deserialize the data type, and the equality operator must be implemented.

10.2 Timing flexibility

It is currently difficult for instructors to have loose timing specifications, since *evaluation points* inherently have a pre-determined time interval. The only flexibility is picking a condition to which that interval is relative. In the next version of MicroGrader, the data structure for test cases will be augmented such that an instructor can give different sorts of timing specification. For example, an instructor can specify that a certain sequence of outputs must occur in a certain order, regardless of timing. It will also be possible to put requirements on the length of time for which an output occurs, regardless of exactly when it occurs.

10.3 Editing recordings

As mentioned in section 9.5, a set of tools will be included in the next version of MicroGrader to facilitate some manual editing of recordings of a instructor's correctly implemented system. This will help create test cases that are unambiguous. Ideally, these tools will also help an instructor programmatically edit a recording.

10.4 Dynamic values

Certain “correct” values are not determined at the time the test case is created. For example, the Wikipedia Scroller assignment has an output that involves the current text on Wikipedia.org. The next version of MicroGrader, will include a “place-holder” data type. This placeholder will be a function that runs at evaluation time to generate the correct value. In the Wikipedia Scroller case, the placeholder would fetch the correct text from Wikipedia.

10.5 Second order outputs

As mentioned in section 8.4, it is sometimes desirable to measure an output that is not directly generated by the microcontroller, but rather is some transformation of an output generated by the microcontroller. In the next version of MicroGrader, instructors will have the ability to add a custom physical model to convert the microcontroller’s outputs to a derived output (e.g. converting a motor voltage to the position of an attached wheel).

11 Appendix A: Client-server protocol

All client-server communication occurs over a USB connection. All multi-byte integers are transmitted in a little-endian fashion. In general, we refer to a message from the client to the server as a *request*. Some requests result in server-to-client *response*.

All messages consist of a header and a body. The header contains important information about the request or response, and specifies the length of the body. Thus, our protocol requires no termination byte. A message is considered complete once the expected number of bytes is received. As a result, all message bodies must have the specified length, or further communication will fail.

The basic structure of a request is as follows:

request ::= <request header><request body>

request header ::= <uint8 code><uint32 timestamp><uint16 body length>

The most significant bit of the request code determines whether or not the server should send a response. If the bit is 0, a response should be sent. If the bit is 1, a response should be omitted. The lower seven bits of the code determine the type of request, and therefore, how the request body should be interpreted. The structure of the request body depends on the particular type of request. The timestamp is measured in milliseconds, and the body length is measured in bytes.

The basic structure of a response is as follows:

response ::= <response header><response body>

response header ::= <uint8 code><uint16 body length>

The second-least significant bit of the response code indicates whether or not an error occurred with the most recent request (1 indicates an error occurred, 0 indicates success). The response body for an error response is always empty.

The least significant bit indicates if the current session is complete (1 indicates yes, 0 indicates no). The server disconnects immediately after sending a response indicating the session is complete. Once again, the body length is measured in bytes. Notice that the server does not include a timestamp in the response header. The structure of the response body depends on the type of request to which the response corresponds.

Broadly speaking, there are three kinds of requests: inputs, outputs, and events.

11.1 Inputs

Input requests are typically used to “inject” pre-determined test values, in place of true sensor readings. Alternatively, input requests can be used to record observed sensor values on an instructor’s system, for the purpose of building tests. The following table describes the request bodies for input requests.

code	name	request body
0x20	Digital Read	<uint8 pin>[<uint8 value>]
0x22	Analog Read	<uint8 pin><analog params>[<int32 value>]
0x30	Accelerometer	<analog params>[<int32 x value><int32 y value><int32 z value>]
0x31	Gyroscope	<analog params>[<int32 x value><int32 y value><int32 z value>]
0x32	Magnetometer	<analog params>[<int32 x value><int32 y value><int32 z value>]

If the optional value or values are included, this message is passing recorded sensor readings from the client to the server. In that case, the body of the response (if there is a response) is empty.

If the optional values are not included, this message is requesting test values. In that case, the body of the response (if there is a response), is just the requested values. For digital reads, the response is a single byte containing a 0 or a 1. For analog reads, the response is a 32-bit integer. For the accelerometer, gyroscope, and magnetometer requests, the response is a sequence of three 32-bit bit integers representing the x-value, y-value, and z-value respectively.

11.1.1 Value Encoding

Encoding for digital values is simple: a single byte containing the value 0 or 1. For analog values, however, MicroGrader uses a system by which the client decides how an analog value becomes discretized. In particular, the client specifies the *analog params* which are defined as follows:

analog params ::= <int32 min bin><int32 max bin><int32 min value><int32 max value>
These parameters indicate that the physical quantity in the range (min value, max value) should be mapped to the range (min bin, max bin) for discretization.

To illustrate, let's examine the behavior of the Teensy client for analog read requests. It uses the values 0 and 3300 for min value and max value respectively. These represent the minimum and maximum voltages, in millivolts, that the Teensy's analog-to-digital converter (ADC) can process. For min bin and max bin, the values depend on the Teensy's current analog resolution. If the resolution is 10 bits for example, the Teensy client would use the values 0 and 1023 for min bin and max bin.

Let's suppose the Teensy makes a request for an analog read using these parameters, and the server observes that the true value to "inject" is x. Let b_{min}, b_{max}, v_{min}, and v_{max} represent the min bin, max bin, min value, and max value respectively. The server must convert the value x to a "bin number", y, in accordance with the given parameters:

$$y = \left\lfloor b_{\text{min}} + \frac{x_{\text{bound}} - v_{\text{min}}}{v_{\text{max}} - v_{\text{min}}} * (b_{\text{max}} - b_{\text{min}}) \right\rfloor$$

where

$$x_{\text{bound}} = \min(v_{\text{max}}, \max(v_{\text{min}}, x))$$

The server responds with the value y. This example discussed a true input request for injecting a value. For input "recordings", the value supplied by the client should also be the bin number as described by the above formula (typically this math would be calculated in hardware by the ADC).

11.1.2 Units

The units for analog requests are not specified. There is no built-in unit for the relevant physical quantities (namely voltage, acceleration, angular velocity, and magnetic flux density). The client implementation and the constructed test cases just need to be consistent. For example, the reference Teensy client implementation uses millivolts for voltage, and any test cases written for this client should use values expressed in millivolts.

11.2 Outputs

Output requests aren't really "requests" in the traditional sense. They are used to report outputs to the server. The following table describes the request bodies for output requests.

code	name	request body
0x21	Digital Write	<uint8 pin><uint8 value>
0x23	Analog Write	<uint8 pin><analog params><int32 value>
0x41	Screen	<tile>*

If the client expects a response, the body of the response is empty. The encoding of values for digital writes is the same as for digital reads (a single byte, 0 or 1). The encoding of values for analog writes is the same as for analog reads. Recall that the value reported is a "bin number".

11.2.1 Screen Encoding

Since the built-in screen type is a monochrome display, in which each pixel is on-or-off, our screen is essentially a bitmap. We represent each pixel using a single bit. Our encoding for the screen is inspired by the U8x8 embedded library [5], and it makes it simple for embedded clients to transmit screen information if their display libraries are U8x8 compatible.

[5] https://github.com/olikraus/u8g2/wiki/u8x8setupcpp

We code encode the screen as a collection of 8x8 *tiles*. The entire bitmap for a tile is contained in an unsigned 64-bit integer. The most significant byte of the 64-bit integer encodes the leftmost column of the tile. The next-most significant byte encodes the second-to-leftmost column of the tile, and so forth. Within a 8-pixel column, the most significant bit represents the state (on or off) of the top pixel, the second-most significant bit represents the state of the pixel below that, and so forth. In our terminology, the leftmost column has the coordinate $x = 0$ and the topmost row has the coordinate $y = 0$. Therefore, the point $(0, 0)$ is the top-left corner of a bitmap. When a tile is transmitted over serial, note that this unsigned 64-bit integer, like all our other primitives, are transmitted in a little-endian fashion. So the lowest-order byte, representing the rightmost column of the tile, is transmitted first.

The screen can be considered a grid of non-overlapping tiles (though padding might be necessary so that tiles perfectly cover the entire screen. When transmitting the screen, the tiles are transmitted by row, from top to bottom (lowest y-coordinate to highest y-coordinate. Within each row, tiles are transmitted from right to left (highest x-coordinate to lowest x-coordinate). Note that the Screen request does not specify the shape of the screen (just the overall number of tile). Therefore, it is imperative that, before sending any Screen requests, the client informs the server of the screen shape using a “Screen Init” request (see the section below).

11.3 Events

Event requests are similar to output requests in that they are used to report information to the server, not receive information from the server. Responses to event requests have empty bodies, just as with output requests. The following table describes the request bodies for event requests.

code	name	request body
0x00	Init	(*empty*)
0x01	Print	<uint8 char>*
0x40	Screen Init	<uint8 width><uint8 height>
0x50	GPS Fix	(*empty*)
0x60	Wifi Request	(*empty*)
0x61	Wifi Response	(*empty*)

The Init request is sent when the embedded client initializes. In addition, the Screen Init request is used upon initialization to inform the server of the width and height of the screen, as measured in 8x8 tiles. The Print request is used to send custom text (ASCII-encoded) from the client to the server. This is particularly helpful for debugging, since it is impossible to perform regular debugging via USB serial when MicroGrader is in use.

It is important to inform the server of certain events, especially ones that have highly variable latencies. The GPS Fix request is used to inform the server that the client has acquired a GPS fix. The Wifi Request request is used to inform the server that a internet request has been sent. The Wifi Response request is used to inform the server that a internet response has been received.

12 Appendix B: Client implementation for Teensy

MicroGrader is built to be cross-platform and is should work with a wide variety of microcontrollers. As of now, the client side of the communication protocol is implemented for only one microcontroller: the Teensy, an Arduino variant. The key for any client implementation is to be as invisible to students as possible. In the case of the Teensy implementation, students write implement projects in a `.ino` file, which has access to a built-in functions. In order to properly implement the protocol, I wrote a library that overrides many of these built-in functions to perform the proper communication with the server. The required changes to the student's `.ino` file are minimal.

At its core, the client consists of a communication library that interacts with the server. It provides functionality for other libraries to send messages to the server; the core communication library adds the request header to the message. If the request requires a response, the communication library waits, and processes, that response, and returns its contents. If the request does not require a response, the communication library returns immediately after sending a request.

The Teensy includes built-in functions to send and receive data over USB. The MicroGrader client for Teensy replaces the built-in `Serial` object with a wrapper object. When in *test* mode or *recording* mode, any print statements made using `Serial` have no effect. For cases where a student or instructor wishes to print some text for debugging, the library includes a `debug` function, which sends a Print message in accordance with the MicroGrader protocol. When the client is in *inactive* mode, `Serial` print statements work normally. Therefore, students can use `Serial` print statements when not connected to the MicroGrader server, but they don't have to worry about those statements interfering with MicroGrader communication when actual testing is happening.

The fundamental general purpose I/O functionality (GPIO) is also wrapped. This allows for recording GPIO events and for injecting test inputs from the host. As with `SerialSerial`, the regular GPIO functions work normally when the client is in *inactive* mode.

For input from the inertial measurement unit (IMU), I modified the MPU9250 [6] library for Arduino/Teensy. For output to a display, I modified the U8g2 [7] library for Arduino/Teensy. If a course uses the Teensy, but uses different libraries for IMU input and screen output, the course staff must modify the source code of those libraries. Specifically, those libraries need to request inputs from the MicroGrader server when appropriate, instead of performing real sensor readings, and need to report information to the MicroGrader server when appropriate. In practice, the code for doing this, with the help of the core communication library, is not complicated.

From a student's perspective, very few changes need to be made in order to run the MicroGrader client. He or she needs to install the core library and the modified versions of the third-party libraries. The line `#include "MicroGrader.h"` must be added to the top of the `.ino` file and an initialization function, `MicroGrader.begin()` must be called at the beginning of the Teensy `setup()` routine.

[6] https://github.com/sparkfun/SparkFun_MPU-9250-DMP_Arduino_Library

[7] https://github.com/olikraus/u8g2/wiki