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 paper, we will describe a software tool 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. 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 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.

At the end of this paper, we will deal with a critical usability feature. In particular, it is often prohibitively tedious for instructors to specify the test inputs and expected outputs manually. Therefore, MicroGrader includes a feature for "recording" the operation of an instructor's system, and the use of that recording to programmatically generate test inputs and expected outputs.

2 Prior Work

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

Discuss strengths and weaknesses of the UT Austin approach

Discuss current 6.S08 situation

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 to arbitrary precision by increasing the resolution sufficiently. 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. 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.

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

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 function is the `==` 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 7.

6 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.

6.1 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.

6.2 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.

6.3 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.

6.4 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.

7 Automatic Test Generation

Using CAT-SOOP [TODO: ref] 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.

7.1 Motivating example

Manually specifying the inputs and evaluation points for a test case can be quite tedious. To illustrate this, let's us consider an exercise from the Spring 2017 offering of 6.S08 called "Wikipedia Scroller".

This exercise involves a digital input connected to a push-button switch, a nine-axis inertial measurement unit (though this exercise only requires x-axis acceleration) and a 128x64 pixel monochrome OLED screen. It also requires web connectivity.

7.1.1 Problem description

Initially, the screen output is blank. When the user performs a "long press" of the push-button, the system enters "text entry" mode. By definition, a long press is a button press lasting at least two seconds.

In text entry mode, a user builds up a query string character by character. For the purposes of this exercise, we only use the lower case letters a-z, the digit characters 0-9, and the space character. In text entry mode, the two relevant state variables are `query` (a string) and `next_character` (a single character). Initially, `query` is the empty string, and `next_character` is the space character. The text `query` + `next_character` should be displayed on the OLED (initially, this is just the string `" "`, so the screen is still empty).

In order to change the value of `next_character` the user tilts the device left or right. If it has been at least 150 milliseconds since the last time `next_character` changed, and the tilt is at least 20° to the right, then `next_character` should increment. If it has been at least 150 milliseconds since the last time `next_character` changed, and the tilt is at least 20°, then `next_chararacter` should decrement. For the purposes of incrementing and decrementing, the character order is defined by the string (notice the leading space):

```
" abcdefghijklmnopqrstuvwxyz0123456789"
```

If the user performs a "short press" (i.e. a push-button press less than two seconds long), `next_character` is locked in. Specifically, the following operations are performed:

```
query = query + next_character
next_character = ' ' // The space character
```

If the user performs a long press, the current query string is sent via HTTP to a server-side component of the project (which is outside the scope of MicroGrader). The server-side script returns the first 200 characters of the Wikipedia page with a name matching the original query string. When the embedded system receives the response, it prints it onto the OLED screen. Another long press takes the user back to text entry mode (with the same initial state values as before), and the cycle repeats.

7.1.2 Why manual test case creation is tedious

As a TA evaluating the student's system, I might perform the following procedure to ensure the assignment is implemented correctly:

TODO: describe my process

It would be fairly tedious (though straightforward) 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 automation.

7.2 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.

7.3 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.

7.3.1 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 web 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 4.4.2 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 7.3.3.

7.3.2 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).

7.3.3 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 (e.g. `digitalWrite(LOW)` over and over again) have no effect. The change in the value is all that matters.

In constructing each evaluation point, we can easily determine the channel. The expected value is simply 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 ought to 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.

7.4 Example of an auto-generated test case

TODO Describe test case generated by aforementioned scaffold and log

7.5 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 ought to 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. TODO: describe edge case ambiguity

Describe strategy to disambiguate

Describe re-recording of outputs

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 A: Client-server protocol

All client-server communication occurs over a USB serial 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 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.

10.1 Inputs

TODO: text

code	name	message 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>]

10.2 Outputs

TODO: text

code	name	message body
0x21	Digital Write	<uint8 pin><uint8 value>
0x23	Analog Write	<uint8 pin><analog params><int32 value>
0x41	Screen Full	TODO
0x42	Screen Tile	TODO

10.3 Events

TODO: text

11 Appendix B: Reference client implementation for Teensy

Describe overall

Describe core functionality (i.e. implementation of client protocol)

Describe replacement of built-in stuff

Describe modified libs

Describe necessary / sufficient code changes for the .ino file

12 References