ReMath Milestone 2 Report

Olivia Lucca Fraser and Daniel Kuehn

March 12, 2021

Contents

1 Overview

The following report provides a brief synopsis of our contribution to the ReMath project, under the working title, "REFUSR: Reverse Engineering Functions Using Symbolic Regression", as it pertains to the second agreed-upon milestone: the generation of a data set of Boolean functions and their implementations in executable PLC code.

2 Generating Formal Specifications and Implementations

2.1 Generating Symbolic Expressions

To generate random, arbitrarily complex Boolean functions of n variables, we designed a simple, recursive constructor:

```
function grow(depth,
              max_depth,
              terminals=TERMINALS,
              nonterminals=NONTERMINALS,
              bushiness=0.5)
    nodes = vcat(terminals, nonterminals)
    if depth == max_depth
        return first(rand(terminals))()
    else
        (node, arity) =
            (depth > 0 && rand() > bushiness) ?
            rand(nodes) : rand(nonterminals)
        args = [grow(depth+1,
                     max_depth,
                     terminals,
                     nonterminals,
                     bushiness)
                for _ in 1:arity]
        return node(args...)
    end
end
```

This method will produce a wide range of Boolean functions, with a variety of properties. We designed a second constructor to generate variants of the widely used Boolean multiplexer circuit (MUX), which uses a sequence of n control bits to select between 2^n different inputs. To present a greater challenge for symbolic regression, our n-MUX constructor will shuffle the input and control variables.

```
function mux(ctrl_bits; vars=nothing, shuffle=true)
    num_inputs = 2^ctrl_bits
    needed = num_inputs + ctrl_bits
    ensure_input_variables!(needed)
    vars = isnothing(vars) ? INPUT[1:needed] : vars
    @assert length(vars)  needed "At least $(needed) vars are needed"
    wires = shuffle ? sort(vars, by = _ -> rand()) : vars
    controls = wires[1:ctrl_bits]
    input = wires[(ctrl_bits+1):end]
    m = foldl(&, map(0:(num_inputs-1)) do i
              switches = bits(i, ctrl_bits)
              antecedent = foldl(&, map(zip(switches, controls)) do (s, c)
                                 s == 0 ? !c : c
                                 end)
              consequent = input[i+1]
              # return the material implication
              antecedent  consequent
              end)
    return m, controls, input
end
```

2.2 Compiling Symbolic Expressions to Structured Text (IEC 61131-3)

Compiling ST expressions from the lisp-like symbolic expressions that our `grow()` method generates is trivial. The main complication we faced is this: the ACE 11 has 6 input pins. The Raspberry Pi has 14. If we limit ourselves to searching for Boolean functions with 6, or even 14, free variables, the effort is too trivial to warrant anything more than a brute force search. There are, after all, only $2^1$4 or 16,384 rows in the truth table of a 14-variable Boolean function, and while iterating through these by hand would be tedious, it's the sort of problem that could be solved with a simple for-loop. If we want to explore a domain of functions that's large enough to warrant the use of sophisticated search and learning techniques, we need to consider functions with a larger number of free variables.

To solve this, we designed a bit of boilerplate ST code, which polls the `%IX` pins and reads successive sequences of parameters, storing them in an array. Once the input array is fully populated, we stop reading input variables, and execute the compiled expression, whose free variables are interpreted as references into that array. The result is then written to an output pin, along with a "ready" flag.

2.3 Calling functions on the PLC

Programs compiled for the target PLC can then be called as if they were ordinary (Julia) functions, implemented on the host system, thanks to a functional abstraction layer. This abstraction layer serializes the function parameters, and feeds them to the device in batches of 5, until the aforementioned input array is populated. It then polls the device, until an output is ready, which it then returns to the caller of the wrapper function. The point of this abstraction layer is to let us easily switch between different PLC devices and emulations, facilitating both development and testing.

2.4 Generating Input/Output Data

Once a Boolean function is generated, it can be executed by either evaluating it as a symbolic expression (using the `SymbolicUtils.jl` library), or on the PLC device, using the functional wrapper. If the number of free variables is small, we can simply generate the truth table for this function. But if the function uses a large number of free variables (and is therefore an interesting candidate for symbolic regression and other sophisticated reverse engineering techniques), we can simply sample an arbitrary number of rows from the truth table.

A representative data set, generated in this fashion, can be found here. The generation of additional functions, ST code, and truth table samples, only requires the execution of our generation tool.

3 Programmable Logic Controllers

We are currently targeting two varieties of PLC: Velocio's ACE controllers (specifically, the ACE11 and the ACE1600), and the open source OpenPLC system, running on Raspberry Pi. In each case, the essentials are more or less the same, except for when it comes to the matter of communicating with the device. Velocio's devices can be programmed only using one of two graphical programming languages (both of which fall under the IEC 61131-3 standard): Ladder Logic and a form of Function Block Diagrams. The Velocio IDE, vBuilder, stores these graphical representations on disk as XML files, which can be easily generated or altered by other means. Before sending these programs to the device, vBuilder compiles them, using a closed-source compiler, to a proprietary and undocumented binary format, which resembles the MODBUS protocol in certain respects.

OpenPLC devices, by contrast, can be programmed using not only Ladder Logic or Function Block Diagrams, but the two text-based programming languages described in the IEC 61131-3 standard as well: the assembly-like language called Instruction List (IL), or the Pascal-like language known as Structured Text (ST). Since IL and ST files are devoid of inessential graphical information, and resemble well-known programming paradigms, we have found them to be easier to read, manipulate, and programmatically generate. The OpenPLC editor, moreover, compiles the IEC 61131-3 graphical languages that it supports down to ST before sending the program to the device. There is no additional complication involved in sending ST code generated in other ways to OpenPLC devices.

3.1 ACE11 and the USB protocol

After capturing the data that is sent to the PLC when you compile and run a program from vBuilder its clear that the protocol over the wire is some type of binary, serial protocol.

The protocol structure seems to be as following:

- Preamble in the form of a four byte prefix (`56 ff ff 00`)
- One byte length field, which includes the length of the preamble unlike how MODBUS usually calculates length beginning (and including) from the length byte
- One byte function code/type that denotes which "class" of function that is being called
- One byte function selector that denotes which function is actually called
- Variable length of actual payload data, spanning from one to several tens of bytes

Seeing as it is a proprietary protocol, the information about its structure is very scarce, but there are a few resources that have helped in mapping some of the functions in the protocol including a python script to send commands to the ACE PLCs developed by a well-known Industrial Control Systems (ICS) penetration tester, Justin Searle. Searle also reached out and helped us by providing a bare-bones Wireshark dissector for the Velocio protocol.

It was clear that it was a protocol that resembled MODBUS RTU, which is unsurprising, since MODBUS RTU is typically what is used by a controller to read and write registers and output pins ("coils") on sensors and other field equipment used in industrial control systems.

From the resources about what commands there are for the PLC we can deduce the following:

- function code `0xf1` contains debug controls (play, pause, step functions, reset)
- function code `0xf0` contains the commands to enter/exit debug mode
- function code `0x11` deals with setting output pins on/off
- function code `0x0a` deals with reading input/output bits
- function code `0x09` seems to denote that it contains a human-readable ASCII substring, with function code `0x12` being used to give the file name for the source code that the code was compiled from

From the read functions we can deduce that it counts pins starting from `0x01` and incrementing it by one for each pin, starting with the input pins.

This means that for the ACE11 that has six digital inputs/outputs, `0x01` to `0x06` is the input pins and `0x07` to `0x0c` are the output pins.

To set the output pins, the protocol uses the 18th byte as a bitfield to denote which pin is to be set instead of using a sequential number to denote the pin to be set, like the read instructions do, allowing multiple output pins to be set at the same time.

Another observation regarding the data that was being sent was that at the very end of the data stream, the human-readable labels from vBuilder were sent to the PLC, in 16-byte, space-padded fields.

These labels were most likely sent to the device so that when you develop a human-machine interface (HMI) for the system that the PLC is part of, it can read the labels from the device and giving the HMI developer a hint of what the input/output is used for. That way a HMI developer doesn't have to have a separate document with mappings of what "coil" or register controls contains what type of value and so on.

3.2 ACE1600

In addition to the ACE11, another of the entry-level PLCs that Velocio offers, the ACE1600, brings another interesting thing to the table, that will hopefully ease the programmatical instrumentation of it.

The ACE1600 sports three digital inputs, six digital output and one RS232, which speaks MODBUS RTU. The PLC itself is in the same form factor as the ACE11. The fact that it's configured to act as a MODBUS RTU slave device is what interests us.

MODBUS is a master-slave protocol, where the master reads and writes coils or registers on the slave devices, which is exactly what is needed when programmatically instrumenting a device.

Through the MODBUS protocol, the hope is that the PLC can be connected to the model/AI as a pure black box that the model/AI can give different inputs and then read the outputs to get data for its calculations.

Another thing it brings to the table is that its programmed with the same program as the ACE11, vBuilder, which means that we can compile the same program to both the ACE11 and ACE1600 and use that to help the reverse engineering effort.

That way, the binaries it creates can be compared to see what is the same in both compiled programs and what is not, which will help to disambiguate the static boilerplate from the actual program, and the connections diagram embedded in the binary.

This will hopefully let us use either whichever Velocio PLC as our black box in the long run.

A sufficient understanding of the Velocio binary protocol will also allow us to generate binary Velocio code using the same framework we are currently using to generate ST for OpenPLC devices. This will let more of the actual testing and learning be automated.

3.3 Sending input to the PLCs

The MODBUS protocol allows us to read from the input pins of the devices, and write to both internal registers and output pins, *but it does not allow us to write to input pins.* Since we wish to model functions that map PLC input to output, further ingenuity is required, here. We are therefore designing an Arduino device that will read messages over a serial USB connection, and pass them along as signals to a PLC's input pins.

4 Codebase

The code for this project is entirely open source, and hosted on Github. Our working languages for this project are Julia, ST, and C. It is our intention to repackage various modules of the project for general use, and make them available to the PLC security community once they're ready.

5 Data Set

A representative data set for this project can be found in our Github repository. This data set, at the time of writing, includes:

- 100 randomly generated Boolean expressions, each employing up to 50 free variables, and having a maximum tree-depth of 8

- 100 5-MUX expressions – multiplexers, each of which switches between 32 inputs using 5 control bits, employing 38 free variables in total, bound to randomly selected inputs

- 100 4-MUX expressions – same as above, but restricted to only 16 inputs and 4 control bits, for a total of 20 free variables

  To each of these expressions is associated:

- a Structured Text file that implements said function for the OpenPLC, and

- a CSV file containing 10,000 randomly sampled rows of the Boolean function's truth table