Experimental Plan for REFUSR

Olivia Lucca Fraser, Lauren Moos, Anthony Di Franco,
and Daniel Kuehn of Special Circumstances

December 21, 2020

Contents

1 The Aim and Scope of this Project

Our contribution to the ReMath research initiative will focus on the recovery of Boolean functions from their implementations in programmable logic controllers, with a particular focus on the ACE 11. In the following sections, we will discuss our general experimental strategy, the cyberphysical target we have chosen to focus upon, and our research and development objectives.

2 Strategy

Our initial observation is that this problem, in its most general terms, closely resembles that of *symbolic regression* (SR). Symbolic regression is a generic machine learning technique for discovering mathematical expressions that optimally fit a set of datapoints. It differs from polynomial regression inasmuch as the *form* of the expression is not given in advance (as an n^{th}-degree polynomial, for instance), but emerges through the free recombination and variation of syntactic elements. The mechanism that generates those datapoints—the function f that we aim to approximate when we perform symbolic regression—remains, in this case, a black box. There's no reason why we cannot proceed, on a first pass, in the same fashion when it comes to recovering formally specified functions from their concrete implementations. So long as we indeed possess the implementation, we can execute it so as to generate as many datapoints as we like—and we can do this without "opening the box" and revealing the mechanism itself. Already, we have everything we need in order to tackle the problem as a type of symbolic regression.

But to leave the function black-boxed, or, worse, to treat it merely as a static and limited collection of datapoints rather than a practically inexhaustible generator of the same, is clearly to leave something on the table. There is no reason, after all, why we *must* keep the mechanism concealed. Why not open the black box, and avail ourselves of whatever information we can find there, and use it to constrain and guide our symbolic regression? In 2.2, we will go over the two primary techniques that we will use in order to extract useful information from the target function *f*.

Our primary case study, for this approach, will be the domain of *Boolean functions as implemented on the ACE 11 programmable logic controller*. For a detailed discussion of this device, the reader may consult section 3.

2.1 Symbolic Regression

Our methodological starting point will be to build a genetic programming (GP) system for symbolic regression. We will initialize populations of mathematical expressions, and, on a first pass, treat the abstract syntax trees (ASTs) of these expressions as *genotypes*. We will consider this set of genotypes to be embedded in a spatial structure that we will call their "geography"—a device that will help to guide the course of their evolution. We define a set of *genetic operators* over these genotypes—methods for mutation and sexual recombination (i.e., crossover). We then proceed by iterations: on each turn, a random batch of nearby expressions are to be evaluated for

their "fitness"—which here means their relative adequacy to the unknown function, implemented by the target cyberphysical system. The fittest specimens of each batch, or "tournament", are made to reproduce (through sexual recombination and mutation), and the least fit are culled from the population, their places usurped by the tournament-winners' offspring.

But what do we mean by "adequacy to the unknown function", here? On a first approximation, we will treat a function f merely as a randomly generated set S of input/output pairs: `{(x, f(x)) | for n random variables x}`. Adequacy, here, is just a matter of curve-fitting, or the reduction of the mean square error (MSE) value of the evaluated expression relative to the datapoints given.

Notice that, at this point, we have not yet opened the "black box" function f, but have restricted ourselves to a random sample S, produced in advance. The production of this dataset is a simple matter, since we have the black-boxed implementation of f already in hand. All we need to do is to generate the random inputs, feed those inputs to f, and record the outputs.

2.1.1 Lexicase Selection

Here, we are dealing, essentially, with what Lee Spector and Thomas Helmuth have called an "uncompromising problem",

> a problem for which it is not acceptable for a solution to perform sub-optimally on any one test case in exchange for good performance on others. [HSM15]

The recovery of *precise* mathematical specifications for the functions implemented in a given cyberphysical system is a problem that meets this description. It is a domain in which approximations are of interest only insofar as they provide stepping stones towards our ultimate goal: an exact specification. We may therefore wish to avail ourselves to a form of evolutionary search that Spector and Helmuth have shown to be particularly well-suited to such tasks. Rather than a traditional tournament algorithm, we may employ [HSM15, lexicase selection] in order to select the parents for each new offspring. The lexicase algorithm presented by Spector and Helmuth is as follows:

```
1) Initialize:
   a) Set CANDIDATES to be the entire population.
   b) Set CASES to be a list of all the test cases in
   random order.
```

```
2) Loop:
   a) Set CANDIDATES to be the subset of the current
   CANDIDATES that have exactly the best performance
   of any individual currently in CANDIDATES for the
   first case in CASES.
   b) If CANIDATES contains just a single individual,
   then return it.
   c) If CASES contains just a single test case then
   return a randomly selected individual from CANDIDATES.
   d) Otherwise, remove the first test case from CASES
   and go to Loop.
```

This technique can be easily adapted to our problem, as we will illustrate below, in our discussion of constrained symbolic regression.

2.1.2 Geographically Distributed Populations

A second refinement, which our team has found quite helpful in another GP context, is to modulate the movement of genes through the population by embedding the latter in a spatial structure, and then restricting competition between individuals to neighbourhoods or regions of that space. This provides a means of preserving genetic diversity and a break against premature convergence. It also offers way of further parallelizing the evolutionary algorithm, since sparsely interacting subpopulations can be explored concurrently.

We have found it fruitful to impose two degrees of "geographical" structure on our populations: first, we split the population into a number of "islands", each evolving, more or less independently, on its own processor (see [WRH98] for details). We then set up a means of migration between islands. At some variable frequency, which we may call the `migration_rate`, two randomly chosen individuals from two different islands may swap places.

Second, the local population of each island is embedded in an n-dimensional torus, where n is adjustable. Tournament competition takes place between neighbours on the torus (the tournament catchment radius can be adjusted, as can the probability of an individual competing in a tournament relative to their proximity to the first challenger). This technique is discussed in some detail in section 10.5 of [PLM08], and Lee Spector has shown the benefits that can be gained by embedding the population in as "trivial" a geography as a 1-torus (a circular buffer) in [SK06], as compared to tournament

selection with no spatial constraints.

Where lexicase, rather than tournament, selection is used, geographical proximity to a privileged "challenger" individual can be used to weight the selection of parents from the `CANDIDATES` set, in the event that the `TESTS` set is depleted. We may also, rather that setting `CANDIDATES` to an entire island population, set it to a geographically clustered subpopulation on the island's toroidal surface.

2.1.3 Genomes in Latent Space (GILS)

One limitation of using a pure genetic programming based approach is that mutation and reproduction, and therefore the search space of the genetic program itself, utilize a notion of Levenshtein Distance as a proxy for semantic similarity/difference between two genomes. Since this is only sometimes the case for machine instructions, there is a great deal of brute-force effort which could be constrained by more sophisticated notions of distance and a model capable of learning grammar. Intuitively, it is unlikely that this method would increase the overall accuracy - it is likely it will increase sample efficiency dramatically and reduce the number of epochs for convergence.

To this effect, we propose use the generation of genomes as a way of bootstrapping the aggregation of supervised labeled data for a deep learning Natural Language Processing model with a notion of long-term/short-term memory and a latent space embedding (Machine2Vec) that can serve as either a more robust search space for a "student" genetic program or a downstream solution (in which case the entire architecture becomes on of self-supervision of a language model through the use of genetic programming). Either of these approaches would represent a state of the art hybridization of genetic programming and deep learning.

2.2 Opening the Black Box

The guiding idea, here, is fairly simple: rather than searching for a "silver bullet" capable of solving the mathematical recovery problem in a single stroke, we begin with the modest but well-established technique of recovering unknown functions from input/output samples through symbolic regression, all the while sharpening the focus of our search with successive layers of formal constraints. These constraints can be obtained by "opening the black box", in various ways. At present, it appears that we have at least two, potentially quite rich sources of information to draw on:

1. the probabilistic *property testing* of the Boolean function, which we are free to probe with whichever inputs we choose

2. the *static binary analysis* of the implementation

One of our first technical objectives will be to set up a system for performing these queries, and devising a domain specific language for expressing the properties that these techniques allow us to infer. These property expressions can then be used to constrain the evolutionary search for function specifications adequate to the implementation in question. Two strikingly similar techniques have emerged in the recent evolutionary computation literature for using formally expressed properties to guide symbolic regression: *logic-guided genetic algorithms*, or LGGA, as developed by Ashok, Scott, Wetzel, Panju, and Ganesh [ASW+20], and *constraint-driven genetic programming*, or CDGP, as developed by Błądek and Krawiec [BK19].

2.2.1 Probabilistic Property Testing

Since we have at our disposal not merely a subset of the target function's graph, but the implementation itself, we can employ this implementation as an "oracle" of sorts: we can feed it any input we like and record its output. This gives us everything we need to avail ourselves of the mathematical theory of *probabilistic property testing* (see [RS96], [Bla12], and [Xie18]). Probabilistic property testing is a technique for determining whether or not a function f satisfies, with high probability, a given property P, on the basis of observing the behaviour of $f(x)$ for a relatively small number of inputs \$x\$—or else if f is "far" from every function that satisfies P.

It is possible, in fact, to do better than testing f at points by using techniques from the field of ultra-arithmetic to probe the function's input/output relationship over dense intervals by using arithmetic models with set semantics for the test values. In brief, ultra-arithmetic represents a function's input-output behavior explicitly with an approximating fuction in a tractable basis such as the Chebyshev, Taylor, or Fourier series, and gives rigorous bounds for the error of the approximation. By iteratively refining the region over which the function's input-output relationship is probed, we can obtain a global model of the function in terms of a standard basis with any desired bound on the error. Such a model can be used directly to approximate the function, or the resulting approximating polynomials can be examined, refactored, or truncated to obtain a succinct algebraic form for the original function.

Such a representation of the function under scrutiny can be used to perform efficient search for global extrema of the value of the function, a common desideratum. See [MB05], for example.

There is not, as far as we know, a suitable software library for probabilistic property testing, at present, and so one of our first objectives will be to produce one, which we hope will turn out to be of general interest to the scientific computing community.

2.2.2 Static Binary Analysis

A wealth of information about the structure of f can be obtained through traditional static binary analysis. By constructing and inspecting the *data-flow graph* (DFG) for f, for instance, we are able to determine which parameters contribute to value of f. An inspection of f's *control-flow graph* (CFG) may allow us to assess f's worst-case computational complexity, relative to input. (For a general discussion of these structures and the algorithms by which they may be extracted from a program, see [NNH10].)

The `angr` reverse engineering framework, for Python, provides a number of tools for extracting this type of information from a binary file, and supports the ARM Thumb instruction set used by the ACE 11. Our task, here, then, is to develop a system for translating structural information obtained through `angr`'s battery of binary analysis techniques into predicates that can be exploited by our constraint-driven genetic programming algorithm.

2.3 Constrained Symbolic Regression (CDGP/LGGA)

Błądek and Krawiec's work on "Constraint Guided Genetic Programming" (see [BK19]) and the strikingly similar but seemingly parallel development of "Logic Guided Genetic Algorithms" by Ashok, Scott, Wetzel, Panju, and Ganesh (see [ASW$^+$20]) show us a way of using formally expressed properties to guide and augment the power of symbolic regression. These techniques are independent of the sources by which we come to know those properties, and so are fully compatible with our proposed use of probabilistic property testing and static binary analysis.

For the time being, let us restrict ourselves to CDGP, and illustrate how it might be applied to our approach as it has been outlined above:

1. We begin by creating an initial set of tests $\mathbf{T}$ for this population. These are produced by generating a random set of inputs for f and collecting the outputs.

2. We then, through the means described in the above discussions of static binary analysis and probabilistic property testing, establish a set of formal properties **C** that f is either certain or highly likely to satisfy. These are then expressed as a set of constraints that any candidate must satisfy if it is to be counted as equivalent to f.

3. We then generate an initial population of symbolic expressions, **P**, as described in the section on symbolic regression, each of which expresses a Boolean function of n variables.

4. Next, we perform a form of lexicase selection [HSM15] on **P**: we iterate through a (geographically localized) subpopulation of **P** in random order, and set aside any expressions that either fail to pass some subset of the tests in **T** or else violate a constraint in **C**, until only two members of remain. These are selected as a mating pair.

5. We then enlist an SMT solver (Z3, for instance) to attempt to generate an input that, when passed to our selected candidates, violates one or more of the constraints in **C**. If such an input can be generated, it will be considered a counterexample to our candidate solutions, and will be fed to f in order to generate a new datapoint, which will in turn be appended to **T**.

6. If, for some candidate **x**, no such counterexample can be found, and *all* of the tests in **T** have been passed without any errors, then we are finished: the symbolic expression **x** can be taken as a probable specification for the function implemented in f. We then skip ahead to step 9.

7. So long as this is not the case, we apply one or more genetic operators (crossover, mutation) to the winning candidates, and insert the resulting offspring into **P**, replacing whichever individuals were first eliminated by the lexicase selection process.

8. We then go back to step 4, and repeat the cycle until a perfect solution to every test pair in **T**, and every constraint in **C**, is found.

9. Once a solution has been found, we test it against a fresh battery of input/output pairs (datapoints) generated by f, to better gauge the accuracy of our search. In any inaccuracies are detected, the discriminating tests are appended to **T**, and we go back to step 4. If not, then we consider the search complete.

The symbolic expressions discovered in this fashion may still be a few steps away from providing mathematical summaries that would be useful to a human subject matter expert (SME), owing to the peculiarly "hairy" and irregular character of evolved programs. The automatic simplification of symbolic expressions is a well-researched domain, however [BBK14]. The simplified symbolic expressions arrived at should be essentially human-readable, and meaningfully interpretable by an SME, and they will be accompanied by a concise statement of the formal constraints **C** that have guided the search, and which the final specimens demonstrably satisfy.

3 The ACE 11, our Cyberphysical Target

3.0.1 What is the ACE 11?

The ACE 11 is a small PLC, with 6 digital in/outputs, that runs off either a USB port or a 2-pin 5V power supply. Its 2.5 inches by 2.5 inches by 0.5 inches. It supports Ladder Logic, Flow Chart and Object Oriented programming, and talks Modbus over USB for receiving programs and getting/supplying values to HMIs (Human Machine Interfaces). The digital outputs can handle 3 - 30 VDC, 300 mA and the digital inputs can handle 3 - 30 VDC. The MCU (Microcontroller) in the PLC is a Texas Instrument 32-bit ARM Cortex-M4F, TM4C123H6PM, which runs at 80MHz. It has 256kB Flash memory, 2 kB EEPROM, 32 kB SRAM and two 12-bit ADC modules. It runs in Thumb-2 mode, which means it has a mixed 16/32-bit instruction set. It also features a 16-bit SIMD vector processing unit, six 32-bit timers (that can be split to 12 16-bit timers) and six 64-bit timers (that can be split to 12 32-bit timers) with real-time clock capability. Alongside that it also has a MPU (Memory Protection Unit) and a single-precision capable FPU (Floating-Point Unit).

3.1 Where is the ACE 11 used?

Velocio Networks targets the ACE line of PLC devices towards everything from hobbyists and small start-ups to large companies that need a flexible and cost-effective solution to deploy a PLC controlled system. There is also the Branch line of PLCs that Velocio Networks offers, that is designed to make larger PLC systems easier to accomplish, by making the PLCs into a distributed system, with a master-worker relation between a master device and the rest of the PLCs.

The ACE line of PLCs are specifically made for smaller implementations, where you have a localized process that needs to be controlled by a single PLC that has between 3 - 12 analogue inputs, 3 - 18 digital inputs, 2 - 4 thermal/differential analogue inputs, 3 - 24 digital outputs and 1 - 2 RS232/RS485 connectors. One example is a container company (ColdBox) that makes temperature controlled transport containers, where a ACE PLC was put as the heart of the temperature regulation system. It was responsible both for the actual regulation of the system, but also external communication through a touchscreen and a cellular modem, showing the flexibility of the ACE PLCs.

Its small device footprint makes it ideal for situations where there isn't that much space in control boxes or in the area of the devices the PLC is going to control. They also offer embedded PLCs, for custom hardware projects where you want to integrate a PLC on a custom PCB.

3.2 How is the ACE 11 programmed

The main software used to make the Ladder Logic or Flow Chart programs that is then run on ACE or Branch PLCs is called vBuilder and is provided completely free of charge by Velocio Networks. It has a easy to use interface and a good manual to get started even for a novice. It comes both as 32 and 64-bit program and is compatible with Windows from Windows Vista up to Windows 10. The manual contains 4 examples, 2 for making a Flow Chart program and 2 for making a Ladder Logic program, amongst the standard manual contents that showcases the interface of vBuilder and how you do different things in the UI. A notable feature for both ACE and Branch PLCs is that they support a more, modern Object Oriented Programming approach, where you can code objects and subroutines to be used. This makes it easier to structure the programs and enables easier code reuse.

The programs that are built with vBuilder can either be compiled to a file, that you then provision the PLC with through a USB connection, or you integrate the PLC with vBuilder and run the code interactively. With the interactive option, you can single step, debug and get a overview of your program as it is running on the PLC. You can stop the program any time and look at the current memory and IO state. They also offer a software that is called vFactory, that is aimed towards designing HMIs that visualize the state of the process that the program that is running on a PLC is in. Its a drag-n-drop interface where you choose the type of visualisation you want, drag it to where you want it on a grid and then you configure the properties that it should have, i.e. what tag it should take its data from in

the program its monitoring, what colour the control should have and similar properties. For the graph-like visualisations you can also choose boundries of the value its monitoring, to have it show different colours depending on the value. There is also a companion software called vFactory Viewer if you're only interested in viewing a HMI that has been built with vFactory instead of both viewing and editing it.

Besides the manufacturers' own software, all of their PLCs are also programmable with the different InduSoft software available from Aveva.

As the PLCs speak plain Modbus over USB, they can interface with, and be programmed by, any software or hardware that can access a PLC over Modbus over USB. The manufacturer has a Modbus example that showcases a Visual Studio made form, programmed in C#, that connects to a Velocio PLC to get/set values.

In addition to the free software used to program both the PLC and HMIs, the manufacturer also supplies eleven tutorials to get started with programming their PLCs, three tutorials to get started with making HMIs (mainly targeted at the HMI hardware that they also sell, Command HMI) and five tutorials that shows how to integrate with different motor controls or other equipment like a scale used to weigh things.

3.3 Using the ACE 11 to generate datasets

The ACE 11 will be the main generator for datasets for our algorithms to explore through coding several programs programmed in Ladder Logic, Flow Chart and Object Oriented programming in vBuilder and let the algorithms analyze the binaries and see if they can recover what symbols are in the binaries. vBuilder has the capability of compiling the code and save it in a binary file instead of directly uploading it to a PLC, which makes it easier for us to get our hands on the binaries to analyze. By knowing what instruction set the MCU runs, we can let the algorithms figure out how the instruction set is used to represent for example a timer or switching an output on/off.

That the THUMB-2 instruction set is a mixed 16/32-bit instruction set means that the state space to cover isn't too large, also the fact that its focus is on code-density and thus only includes a subset of the full ARM instruction set means the state space is even more reduced.

That vBuilder has the ability to output the compiled code into binaries means that we can easily generate a large corpus to feed as data to the algorithm to train it. It also means that we don't need to instrument a USB capturing tool to be able to capture the binary as its sent to the PLC for execution.

We aim to be able to both dissect the binaries and get a understanding of how the PLC programming language uses the Cortex-M4F to run its programs and be able to analyze the PLC while running the code and see if the algorithms can recover what is being executed in terms of symbols. PLC languages are usually fairly bit-oriented and thus can be approached like boolean algebraic equations in most parts. Language features like timers and counters are important for the logic of a program, but don't neccessarily fit well into boolean algebra, which will be a challenge to tackle.

The generated binaries will also be used to manually reverse engineer the symbol to machine code relation to see if there is anything that can be found regarding relations between type of symbol and the type of instructions used by the compiler to execute that symbol. A big difference between the ARM instruction set and the Thumb-2 instruction set is that almost all instructions in Thumb-2 are unconditional and instead Thumb-2 have a special If-Then instruction to use to make conditionals. This reduces the complexity of reverse engineering the machine code.

4 Software Development Objectives

We have decided to use Julia as our primary development language for this project, owing to its interactive nature, its interoperability with Python and C, its runtime efficiency, its mature scientific computing ecosystem, and its familiarity to our research team.

This project will involve the development of the following resources, in the Julia programming language, which we hope to be of general interest to the scientific programming community:

1. A probabilistic property testing Library
2. An API for interacting with the ACE 11
3. A static binary analysis toolbox, providing an ergonomic interface to the Python `angr` framework for reverse engineering
4. A tree-based genetic programming library, with support for GILS and CDGP.
5. A report generation tool, that will simplify symbolic expressions resulting from A, display them in a human readable format, and document the general properties and constraints that f was found to have, producing a document that will be of use to the SME and reverse engineer.

References

[ASW+20] Dhananjay Ashok, Joseph Scott, Sebastian Wetzel, Maysum Panju, and Vijay Ganesh. Logic guided genetic algorithms, 2020.

[BBK14] David H. Bailey, Jonathan M. Borwein, and Alexander D. Kaiser. Automated simplification of large symbolic expressions. *Journal of Symbolic Computation*, 60:120 – 136, 2014.

[BK19] Iwo Błądek and Krzysztof Krawiec. Solving symbolic regression problems with formal constraints. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '19, page 977–984, New York, NY, USA, 2019. Association for Computing Machinery.

[Bla12] Eric Blais. *Testing properties of Boolean functions*. PhD thesis, Canegie Mellon, 1 2012.

[HSM15] T. Helmuth, L. Spector, and J. Matheson. Solving uncompromising problems with lexicase selection. *IEEE Transactions on Evolutionary Computation*, 19(5):630–643, Oct 2015.

[MB05] K. Makino and M. Berz. Verified global optimization with taylor model based range bounders. *WSEAS Transactions on Computers archive*, 4:1611–1618, 2005.

[NNH10] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Publishing Company, Incorporated, 2010.

[PLM08] Riccardo Poli, William Langdon, and Nicholas Mcphee. *A Field Guide to Genetic Programming*. 01 2008.

[RS96] Ronitt Rubinfeld and Madhu Sudan. Robust characterizations of polynomials withapplications to program testing. *SIAM J. Comput.*, 25(2):252–271, February 1996.

[SK06] Lee Spector and Jon Klein. *Trivial Geography in Genetic Programming*, pages 109–123. Springer US, Boston, MA, 2006.

[WRH98] Darrell Whitley, Soraya Rana, and Robert Heckendorn. The island model genetic algorithm: On separability, population size and convergence. *Journal of Computing and Information Technology*, 7, 12 1998.

[Xie18] Jinyu Xie. *Property Testing of Boolean Functions*. PhD thesis, Columbia, 6 2018.