Reverse Engineering Functions Using Symbolic Regression (REFUSR)

Olivia Lucca Fraser (Special Circumstances)

2021-01-26

Outline

Statement of the Problem

Given a cyberphysical implementation (CPI) of an unknown Boolean function, find a symbolic specification of that function.

Our Strategy

- If we consider the CPI as a black box, then, so long as we can execute the CPI, and generate a set of data points, then this looks like a problem of symbolic regression.
- Even as a black box, it has an advantage over a static set of data points: it can be queried as an oracle. So we can probe it with crafted inputs, so as to better infer its properties (probabilistic property testing).
- But the CPI is not a black box! We can open it up and subject it to static program analysis, inferring a second set of properties.
- We can then use these inferred properties to constrain the symbolic regression. One technique for doing so is known as constraint guided genetic programming (CDGP).

Cyberphysical Target: The ACE 11 PLC

Cyberphysical Target: The ACE 11 PLC

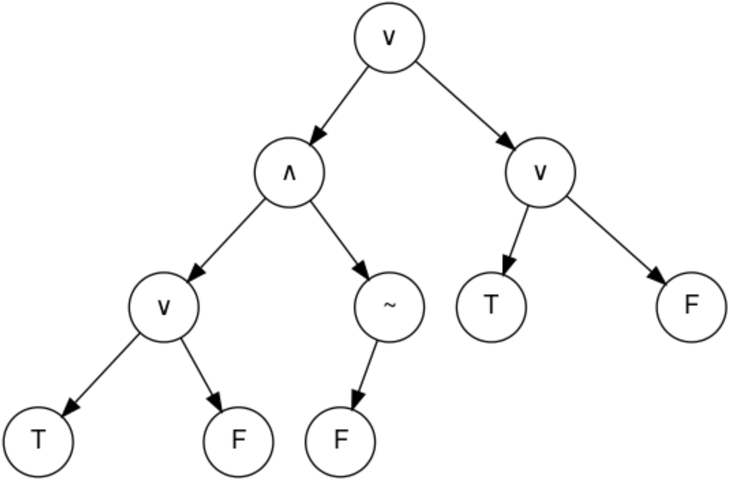
- Small, inexpensive programmable logic controller
- Programmable by Modbus over USB
- Microcontroller is a TI 32-bit ARM Cortex-M4F (TM4C123H6PM)
- Runs in Thumb-2 mode
- 256kB flash memory, 2kB EEPROM, 32kB SRAM
- 16-bit SIMD vector processing unit
- multiple timers
- single-precision floating-point unit

Domain

In the initial phase of our research, we are restricting our focus to Boolean functions, as implemented on the ACE 11 programmable logic controller.

Boolean Symbolic Expressions as Genotypes

We are searching for human-readable, formal specifications of Boolean functions, in the form of symbolic expression trees.



So we initialize a population of random Boolean expressions.

Symbolic Regression through Genetic Programming

Our methodological starting point will be to build a genetic programming (GP) system for symbolic regression.

- initialize a random population of function specifications (genotypes)
- embed this population in a spatial or "geographical" structure
- iteratively evaluate random neighbourhoods of genotypes, in tournaments
- assign fitness relative to how well a genotype's evaluation fits the available datapoints
- cull the least fit of each batch
- reproduce the most fit, through such genetic operators as crossover and mutation

An Alternative to Tournament Selection: Lexicase Selection

Here, we are dealing 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.*

Helmuth and Spector have shown that for many such problems, their technique of lexicase selection outperforms tournament selection.

Lexicase Selection

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.

Opening the Black Box

So far, we have been treating the problem of recovering symbolic mathematical specifications from implementations as a black box problem, where all we have at our disposal is a set of the black box's inputs and outputs.

But in doing so we leave a tremendous amount of information on the table.

How can we put this additional information to use?

Constraint-Driven Genetic Programming (CDGP)

- Begin by creating an initial set of tests T that any solution must pass: generate a random set of inputs for f and collecting the outputs.
- Establish a set of formal properties C that f is either certain or highly likely to satisfy.
- Generate an initial population of symbolic expressions, P, each of which expresses a Boolean function of n variables.

Constraint-Driven Genetic Programming (cont.)

- Perform tournament or lexicase selection on P, to select a mating pair.
- 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 be appended to T.

Constraint-Driven Genetic Programming (cont.)

- 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*.
- 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.

Constraint-Driven Genetic Programming (cont.)

- We repeat the process until a solution is found.
- 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 resume the search. If not, we consider the search complete.

But where do our constraining properties come from?

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". We can feed it any input we like and record its output.
- This gives us all we need to avail ourselves of the mathematical theory of *probabilistic property testing*.
- This 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.
- We plan on developing a Julia library that can be used to perform PPT on arbitrary black-boxed functions, or oracles.

Static Program 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.

Summary

- Our approach to formula recovery goes from outside to inside, rather than the reverse.
- We begin with a black box, and then progressively constrain the search for symbolic specification by bringing constraints into play.
- These constraints may come from heterogeneous sources: probabilistic property testing, static analysis, and perhaps others can be added as we go.